NADKARNI, ADWAIT PRAVIN. Towards Practical Data Secrecy in Modern Operating Systems. (Under the direction of Dr. William Enck.)

Modern commodity operating systems such as Android, iOS, and Windows 8 have changed the way consumers interact with computing devices. On these operating systems, third-party applications provide tremendous functionality for creating, processing, managing, and sharing user data. Moreover, users can complete complex tasks by stringing together purpose-specific third-party applications (e.g., an email application, a document viewer, a barcode scanner). However, while user-directed information sharing among applications provides value, it exposes user data to the risk of accidental or malicious exfiltration from the device.

In this dissertation, we explore the design, enforcement and specification of security policies for preventing unauthorized disclosure of user data. First, we consider the policy expressibility requirements of applications for protecting their application-specific data. We demonstrate the feasibility of our policies for preventing accidental disclosure of data through the design and implementation of the Aquifer policy framework for Android. Second, we describe how information flow control enforcement can be made both secure as well as practical, by defining the primitive of lazy polyinstantiation. We design and implement the Weir system, which integrates lazy polyinstantiation into Android, and demonstrates practical and secure information flow control enforcement with unmodified applications. Finally, we motivate the need for user-specific security policy specification, and describe a novel approach for predicting security policies based on examples. The lessons learned from these techniques motivate the treatment of user-directed sharing as a first-class security problem, and lay the groundwork for involving the user in specifying user data security policies.
Towards Practical Data Secrecy in Modern Operating Systems

by
Adwait Pravin Nadkarni

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2017

APPROVED BY:

__________________________________________
Dr. Douglas Reeves

__________________________________________
Dr. Munindar Singh

__________________________________________
Dr. Jessica Staddon

__________________________________________
Dr. William Enck
Chair of Advisory Committee
DEDICATION

To my mother and father, who cultivated in me the love of learning.
BIOGRAPHY

The author was born and raised in Mumbai, India. Prior to joining the doctoral program at the North Carolina State University, the author earned his Bachelor of Engineering (BE) in Computer Engineering from the University of Mumbai in July 2011, followed by a Master of Science (MS) in Computer Science from North Carolina State University in December 2012. The author was a founding member of the Wolfpack Security and Privacy Research (WSPR) Lab at NC State, and served as its Lead Graduate Student from August 2015 to May 2017.
ACKNOWLEDGEMENTS

This dissertation is the culmination of 6 years of my education at the North Carolina State University (NC State). My experiences at NC State, whether with the faculty, students, staff or the institution itself, have played an important role in my growth as a professional and as a person. I would like to use this opportunity to thank all the people that have contributed to my life and career.

I would like to thank my advisor, Dr. William Enck, for his guidance and support over the course of my time in graduate school. Will's class on computer security was the first class I took at NC State. His teaching and philosophy are what inspired me to pursue research in computer security and an academic career. Over the years, Will helped me mature as an independent researcher with his unconditional support and unbiased advice. He taught me to always put students first, a philosophy that has now become an inseparable part of my personality as a mentor. I owe the successful completion of my PhD entirely to Will's mentoring, dedication, and friendship.

Over the course of my PhD, I have had the benefit of interacting with fellow researchers who have helped shape my research with their insights and broad perspectives. I would like to thank my collaborators, specifically Dr. Somesh Jha, Dr. Ahmad-Reza Sadeghi, Dr. Jessica Staddon, Dr. Xiaohui Gu, Dr. Tao Xie, Dr. Stephan Heuser, Dr. Anmol Sheth, Dr. Nina Taft, Dr. Udi Weinsberg, Benjamin Andow and Rui Shi for their efforts and guidance. I would like to thank Michael Grace for mentoring me during my internship at Samsung Research America, and for giving me the freedom to explore a broad array of research problems during the course of my internship. I would like to thank members of my doctoral committee, Dr. Douglas Reeves, Dr. Munindar Singh, and Dr. Jessica Staddon for their valuable feedback during the course of my degree. I would also like to thank Dr. Somesh Jha, in particular, for mentoring me during the last couple of years of my PhD, and guiding me through to the next step of my career.

A significant portion of my research has been inspired by interactions with student members in the Wolfpack Security and Privacy Research (WSPR) Lab. Being a part of and leading the WSPR lab for the last two years has been an unforgettable experience. I would like to thank all my labmates for their invaluable support and expertise, specifically, Benjamin Andow, Luke Deshotels, Sigmund Gorski, Micah Bushouse, Russell Meredith, Isaac Polinsky, Akash Verma, Andrew McNamara, Sanghyun Ahn. I would also like to thank
past members of the security lab at NC State, specifically, Jitesh Shah, Vasant Tendulkar, Ashwin Shashidharan, Dr. Jason Gionta and Zach Jorgensen, for their feedback during my early years at NC State.

I would like to thank my family for their love, support and patience. My mother and father always encouraged curiosity and independent thought and taught me to see things through to the end. These qualities have helped me develop as a scientist, for which I am eternally thankful to my parents. I would like to thank my two cats for being a constant source of entertainment. Finally, I would like to thank my wife, Radhika, for being my best friend and companion, but most importantly, for always believing in me.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................. x

LIST OF FIGURES ............................................................... xi

Chapter 1 Introduction ...................................................... 1
  1.1 Thesis Statement ....................................................... 4
  1.2 Contributions .......................................................... 7
  1.3 Dissertation Outline ................................................ 9

Chapter 2 Background ....................................................... 11
  2.1 Modern Operating Systems ......................................... 12
    2.1.1 Applications as Security Principals ......................... 12
    2.1.2 User-directed Information Sharing ......................... 13
    2.1.3 User Data Secrecy in Modern Operating Systems ........ 13
  2.2 Android ................................................................. 14
    2.2.1 Architectural Overview ....................................... 15
    2.2.2 Application Model ............................................. 16
    2.2.3 Inter-Component Communication ............................. 17
    2.2.4 Android's Security Architecture ......................... 18
  2.3 DIFC Fundamentals .................................................. 19

Chapter 3 Related Work .................................................. 22
  3.1 Operating Systems Security ........................................ 22
    3.1.1 Discretionary and Mandatory Access Control ............. 24
    3.1.2 Information Flow Control (IFC) ........................... 24
    3.1.3 Decentralized Information Flow Control (DIFC) ........ 26
    3.1.4 DIFC on Android .............................................. 30
  3.2 Modern OS Security .................................................. 31
    3.2.1 Threats to User Data ........................................ 31
    3.2.2 Defense of User and Application Data .................... 34
  3.3 Policy Specification ................................................ 40
    3.3.1 Automatic customization of privacy policies .......... 40
    3.3.2 Languages for expressing security policies ............ 41
    3.3.3 Enforcement of security policies ........................ 42
    3.3.4 Usability of policy interfaces ............................ 42

Chapter 4 A Programmable Framework for Extending Android Security .... 44
  4.1 Design Goals .......................................................... 46
  4.2 Authorization Hook Semantics .................................... 47
  4.3 Architectural Overview ............................................. 49
7.6.2 Policy Change ................................................. 151
7.6.3 Unconfirmed Policy Change .............................. 152
7.6.4 Tag Confusion ............................................. 152
7.7 Lessons ....................................................... 153
7.8 Threats to Validity .......................................... 154
7.9 Summary ..................................................... 155

Chapter 8 Directions for Data Security in Modern Commodity Platforms ........ 156
8.1 The Inevitability of User-directed Sharing ..................... 157
8.2 Future Directions in User-driven Policy Specification ........... 158
  8.2.1 Quantifying the information needed from the user .......... 158
  8.2.2 Acquiring policies from non-experts ...................... 159
  8.2.3 Identifying and providing the contextual information needed by the user .................................................. 160
  8.2.4 Accounting for change in the user's policy ................ 161
8.3 Security of User Resources in Emerging Environments ........ 162
  8.3.1 Automation, and the Absence of UI Interaction ............ 162
  8.3.2 Network Export and Declassification ...................... 163
  8.3.3 Integrity Requirements .................................. 164
8.4 Concluding Remarks .......................................... 164

BIBLIOGRAPHY ..................................................... 166

APPENDICES ...................................................... 185
Appendix A Aquifer Policy Safety Proof ................................ 186
Appendix B Weir Appendix ........................................ 190
  B.1 Weir Component Polyinstantiation Logic ....................... 190
  B.2 Weir DIFC Policy .......................................... 192
  B.3 Application List for Compatibility Evaluation ................ 193
Appendix C Policy By Example (PyBE) Appendix ..................... 195
  C.1 Tags used in the User Study ................................ 195
  C.2 Collecting Examples ....................................... 195
  C.3 Collecting Weights ........................................ 196
  C.3.1 Grouping tags ......................................... 196
  C.3.2 Ordering Groups ....................................... 197
  C.4 Review of Examples ......................................... 198
  C.4.1 Manual Review .......................................... 198
  C.4.2 Semi-automatic Review .................................. 199
  C.5 Testing with Random Examples ............................... 200
    C.5.1 Labeling Test Samples ................................. 200
    C.5.2 Post-test Review ...................................... 200
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Classification of authorization hook semantics required by Android security enhancements</td>
<td>47</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Application Survey Results</td>
<td>74</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Microbenchmark Results</td>
<td>75</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Performance - Unmodified Android (AOSP), Weir in unlabeled context, Weir in labeled context.</td>
<td>112</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Bob's example policies for the WorkCloud policy.</td>
<td>126</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Bob's extended set of examples for the WorkCloud policy target, with newly added examples in <strong>bold</strong>.</td>
<td>128</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>The policy targets used in our study, along with the action they control.</td>
<td>138</td>
</tr>
<tr>
<td>Table 7.4</td>
<td>Number of unique example scenarios created by each user, as well as the total number of policy examples created after assigning decisions for 5 policies.</td>
<td>144</td>
</tr>
<tr>
<td>Table 7.5</td>
<td>Number of random policy scenarios created for testing predictions for each user. Policy decisions are acquired from the user and predicted by PyBE for 5 policies.</td>
<td>144</td>
</tr>
<tr>
<td>Table 7.6</td>
<td>Accuracy of PyBE in comparison with the CoinFlip (abbreviated to CF) baseline, for all 40 user-policy cases. Cases where the accuracy of CF is greater are highlighted in <strong>bold</strong>.</td>
<td>145</td>
</tr>
<tr>
<td>Table 7.7</td>
<td>Accuracy of PyBE in comparison with the MostFreq (abbreviated to MF) approach, for all 40 user-policy cases. Cases where the accuracy of MF is greater are highlighted in <strong>bold</strong>.</td>
<td>145</td>
</tr>
<tr>
<td>Table 7.8</td>
<td>The number of errors identified via manual review of examples (abbreviated as MR), and the additional errors found in the PyBE-assisted review. Cases where the manual review finds more errors are highlighted in <strong>bold</strong>.</td>
<td>147</td>
</tr>
<tr>
<td>Table 7.9</td>
<td>Breakdown of incorrect predictions into misconfigured weights (W), policy change (C), unconfirmed policy change (U) and label confusion (L), across all participants and policies.</td>
<td>148</td>
</tr>
<tr>
<td>Table 7.10</td>
<td>A subset of policy examples initially specified by P1, which includes only those examples that contain Work.</td>
<td>149</td>
</tr>
<tr>
<td>Table B.1</td>
<td>Package names of singleTop, singleInstance and singleTask applications analyzed for compatibility.</td>
<td>194</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.1</td>
<td>Architecture of the Android operating system</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>ASM framework architecture</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>ASM Hook Invocation</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>A lifecycle hook that allows modified data to be returned to the receiving application.</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>The getDeviceId Hook</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Third Party Hooks</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Document signing use case with four apps. A confidential contract received via Email is 1) read in a viewer, 2) converted to PDF, 3) embedded with a written signature, and 4) emailed back to the sender.</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Aquifer policy abstraction</td>
<td>61</td>
</tr>
<tr>
<td>5.3</td>
<td>Aquifer architecture for Android</td>
<td>61</td>
</tr>
<tr>
<td>5.4</td>
<td>Aquifer File Label Propagation Time. Error bars indicate 95% confidence intervals</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Aquifer policy modifications to K-9E Mail</td>
<td>77</td>
</tr>
<tr>
<td>5.6</td>
<td>Aquifer Workflow Filter, a) Without policy and b) With Policy that allows only K-9E Mail</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Shared state in memory: Instances of the WPS Office application performing different tasks in the same process.</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Overview of floating labels w/ polyinstantiation relative to explicit and floating labels.</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Illustration of Weir's lazy polyinstantiation, with three app components: activities A and B, and a service C</td>
<td>96</td>
</tr>
<tr>
<td>6.4</td>
<td>Weir's storage polyinstantiation using layers.</td>
<td>99</td>
</tr>
<tr>
<td>6.5</td>
<td>OfficeEmail creates and adds a tag to an intent</td>
<td>107</td>
</tr>
<tr>
<td>6.6</td>
<td>Floating label DIFC system: Q receives 1 and guesses 0 for every reply not received.</td>
<td>108</td>
</tr>
<tr>
<td>6.7</td>
<td>Weir: Q always receives data “11”</td>
<td>109</td>
</tr>
<tr>
<td>6.8</td>
<td>Time to start a component when 0 → 100 instances (in steps of 10) already exist.</td>
<td>114</td>
</tr>
<tr>
<td>6.9</td>
<td>BCloud’s policy configuration</td>
<td>115</td>
</tr>
<tr>
<td>6.10</td>
<td>BCloud raises its label</td>
<td>115</td>
</tr>
<tr>
<td>6.11</td>
<td>BCloud starts itself with a new label</td>
<td>116</td>
</tr>
<tr>
<td>7.1</td>
<td>An overview of the PyBE approach</td>
<td>125</td>
</tr>
<tr>
<td>7.2</td>
<td>{Home, Memo} disagrees with the majority policy decision of its nearest neighbors.</td>
<td>129</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.3</td>
<td>There is no majority policy decision among the nearest neighbors of {Home, Photo}.</td>
<td>130</td>
</tr>
<tr>
<td>7.4</td>
<td>NN graph for vertex ( p ), consisting of neighbors ( q ) and ( r ) at the closest distance ( d ), violates Inv-1 and Inv-2.</td>
<td>136</td>
</tr>
<tr>
<td>7.5</td>
<td>Screenshot of ( P_1 )'s partial order over tag groups.</td>
<td>141</td>
</tr>
<tr>
<td>7.6</td>
<td>A suggestion made by the PyBE algorithm during the interactive review process.</td>
<td>142</td>
</tr>
<tr>
<td>B.1</td>
<td>Flow of the Activity Manager starting a component. The areas modified or added by ( Weir ) are shaded.</td>
<td>191</td>
</tr>
<tr>
<td>C.1</td>
<td>Tags used in the user study, and the number of users (out of 8) that use each tag.</td>
<td>196</td>
</tr>
<tr>
<td>C.2</td>
<td>Screenshot of the tag groups customized by ( P_1 ).</td>
<td>197</td>
</tr>
</tbody>
</table>
Smartphones and tablets have dominated the commodity computing discourse for the last decade. As of 2016, more than a half of the world’s adult population owns a smartphone. This number is projected to rise to more than 80% by 2020 [Eco15]. The dramatic rise in the use of mobile devices can be attributed to their advanced computing capabilities, combined with nearly unlimited mobility. Today, smartphones and tablets have replaced traditional computing devices (e.g., laptops, desktops) as the key mediums of Internet access [Smi15].

Mobile devices have also found use in the enterprise. Although a handful of demanding computing activities (e.g., programming) may still require laptops or desktops, mobile devices simplify many core tasks such as communication (e.g., Email). Many enterprises now allow employees to use their own device under a bring your own device (BYOD) policy, which regulates how employees may use personal devices to manage work data, while maintaining the security requirements of both. Since its introduction in 2009, over 70% organizations support BYOD for some or all of their employees [Ham15]. The pace at which
smartphones and tablets have replaced traditional computing devices at home and work is nothing short of a revolution.

The primary drivers of this revolution are mobile applications, colloquially known as apps. While the mobile device connects users to the Internet on the go, apps empower the user to exploit the Internet in every way conceivable. The catch-phrase “there's an app for that” correctly captures the diversity and endless potential of applications. Applications connect users to social networks (e.g., Facebook, Twitter), increase productivity by facilitating communication (e.g., Gmail, WhatsApp) or document editing on the go (e.g., MS Word, Google Docs), or even perform tasks that have traditionally required separate office equipment, such as using a mobile device's camera to transform it into a hand-held document scanner (e.g., CamScanner, TinyScanner). As of December 2016, the two mobile platforms with the most market share, i.e., Android and iOS [IDC16], also have the largest application stores, with each offering more than 2 million applications [Sta16]. The availability of a large array of applications may be imperative for the success of a mobile computing platform.

For promoting applications, a mobile platform requires an operating system (OS) that can seamlessly install and run third party applications. Applications often need to access various user and device resources to fulfill their purpose; e.g., location-based applications need GPS location (e.g., Google Maps, Groupon), file managers need storage access (e.g., ES File Explorer, Root Explorer), scanners and camera applications need access to the camera (e.g., Camscanner, Open Camera). The task of providing applications access to security and privacy-sensitive resources, while securing the resources against malicious applications, falls squarely on the OS.

OS security architectures have evolved to address the security challenges of app-based environments. Modern commodity operating systems such as Android, iOS and Windows Phone treat applications as first-class security principals [Roe12; Wan07], providing a significant advantage over access control on traditional OSes such as Linux, Windows 7 or Mac OS X. In the traditional setting, an application executes with the identity (and the privileges) of the user that executes it. As a result, the application can access all of the user’s resources without explicit authorization from the user. This problem of ambient authority facilitates malicious software such as Trojans, i.e., programs that masquerade as useful software but exploit the rights of the program’s user in a way that the user does not intend [Lan94]. On the contrary, when applications are security principals, they can be
issued a subset of the user's privileges (e.g., GPS, camera, SD Card access), thereby limiting the potential damage caused by a Trojan; e.g., while a Trojan executed by a user on Linux can steal all of the user's data, a Trojan on Android can only access data explicitly authorized by the user either at install-time or run-time.

As security principals, applications get ownership of the data they bring to the device. The OS provides applications with private data stores that are only accessible by the respective applications, i.e., data owners (e.g., private application directories on Android and iOS [Dev; Inc]). However, this isolation of data is not rigid. When the user wants to combine the data and functions of multiple applications, the user can direct applications to explicitly share data. For example, when the user chooses to view an email attachment (e.g., a PDF file), the email application shares the attachment with the document viewer application chosen by the user. User-directed information sharing enables the user to accomplish complex tasks with purpose-specific applications.

The utility of user-directed data sharing is accompanied by the risk of unauthorized data disclosure, as third party applications may not be implicitly trusted with the data of other applications. The security enforcement for application data on modern commodity OSes is non-transitive, i.e., once the data owner shares data with another application, it loses control over the flow or use of its information. For example, the email application cannot prevent a malicious or compromised document viewer from exfiltrating the user's attachments. The problem of unauthorized disclosure of user data is particularly relevant to modern commodity OSes, where applications can control access to their own data by the virtue of being security principals, but must give up that control at the user's direction.

This dissertation aims to develop practical policy and enforcement primitives for protecting the user's application-specific data from unauthorized disclosure, without adversely affecting application inter-operability and data sharing. It is important to note that while all modern commodity OSes support user-directed sharing (e.g., Windows Share Charms [Mic13], iOS AirDrop [App13]), Android deserves exclusive focus as it makes flexible inter-application communication and application inter-operability a central design objective [And16a]. In this dissertation, we choose Android as the target of our investigation, primarily due to its advanced primitives of application inter-operability, popularity, and the availability of source code.
1.1 Thesis Statement

Smartphone applications are generally purpose-specific, which requires applications to cooperate for performing complex tasks; e.g., an email application does not implement a “viewer” for attachments, instead outsourcing the functionality to a viewer selected by the user. Inter-application communication primitives in the OS facilitate such inter-operability, while user-directed sharing allows users to dynamically choose the best application to process their data. This dissertation addresses the challenge of providing protection against the unauthorized disclosure of user data while supporting seamless application integration.

Purpose-specific applications often develop application-specific user data objects that capture the abstractions required by their core purpose. As app-specific data objects such as notes, whiteboard snapshots and scanned documents are stored and processed using conventional OS data objects (e.g., files), the OS is oblivious to their existence or properties. That is, app-specific user data is abstract from the OS’s perspective; its security requirements (i.e., the security context) is only known to the application developer or the user. Thus, access control policies defined by a system designer or administrator are insufficient for protecting app-specific data, as the designer lacks knowledge of the security context.

Modern commodity operating systems expose nominal protection mechanisms to applications. For instance, Android’s permissions are the main line of defense for protecting application-specific user data. Each application is assigned a certain set of permissions, which are capabilities that control access to user data and resources. Applications can define and use permissions to control access to their own application-specific data. However, permissions only provide protection at the point of access, and not security for the information accessed. That is, permissions do not apply to copies of data made after an initial access, and the data owner loses all control over the flow of its data once data is shared with other application. The application receiving the data can export it to an adversary on the network, without authorization from the user or the data owning application. Unauthorized disclosure is an information flow problem that permissions or other non-transitive mechanisms are not designed to solve.

Information flow control (IFC) can guarantee data security, and especially protection against unauthorized disclosure [Den76]. Unfortunately, the benefits of IFC have historically
been limited to restricted scenarios such as the militarily, mainly due to the practicality challenges of integrating IFC with commodity OS architectures. More importantly, IFC systems require a centrally mandated policy, which is insufficient for expressing the security requirements of abstract user data on commodity OSes.

Prior work has proposed the use of decentralized information flow control (DIFC) [ML97; ML00] to allow mutually mistrustful security principals to define the security policy for their own data. Taking advantage of the fact that applications are security principals on modern commodity OSes, this dissertation demonstrates how applications acting as data owners may specify security policies for their application-specific data. As described previously, we choose Android as the target platform for our analysis and experimentation.

In this dissertation, we study the challenges and opportunities for integrating practical data secrecy guarantees for application-specific user data in modern OSes, specifically Android. First, we design DIFC policies that allow applications to express the secrecy requirements for their data, specifically the network export of their data by other applications on the device [NE13]. While our policy design accounts for an adversarial setting, our enforcement is relaxed for backwards compatibility with unmodified third party applications. Second, we analyze the challenges in providing practical and sound primitives to enforce DIFC policies in the presence of a malicious adversary [Nad16]. We design the primitive of lazy polyinstantiation to enable sound enforcement without modifying legacy applications. Third and finally, we identify the lack of practical policy specification for abstract user data. We describe the approach of Policy by Example (PyBE) (in submission) to specify user-specific security policies for application-specific user data.

From our analysis of modern commodity operating systems, we observe that the unique abstractions in modern OSes such as Android provide opportunities to integrate practical and secure information flow control guarantees for user data. This observation leads us to the thesis statement of this work, as follows:

Modern operating systems not only expose new secrecy risks for user data, but also provide unique abstractions that enable practical and secure information secrecy guarantees.

Evidence for this thesis can be found throughout this dissertation. Our first study demonstrates that a diverse array of new security challenges on modern commodity operating
systems may be addressed using a common set of authorization hooks in the operating system. Our second study motivates the enforcement of secrecy guarantees for application-specific user data. We design DIFC policies that control the export of information, and demonstrate how Android’s component model and user-directed sharing abstractions facilitate the design of complex policies that rely on conditions on application chains created via user-directed sharing. Our third study leverages Android’s indirect inter-component communication for practical and sound DIFC enforcement, i.e., without relaxing security guarantees or breaking existing applications. Our final study demonstrates the feasibility of user-specific policy specification for application data using organizational tags provided by applications as the building blocks of user-specific policies.

Our results motivate future directions in the security of user data on emerging commodity platforms. First, we drive future security research to consider user-directed inter-application sharing as an indispensable aspect of emerging commodity platforms. Doing so requires keeping the user in the loop when forming the security policy. The Policy by Example (PyBE) approach proposed in this dissertation provides a feasible starting point for user-specific policy specification. However, a general and complete treatment of the problem requires addressing four specific research problems, i.e., (1) Quantifying the minimum fraction of access control information that must be acquired from the user to sufficiently supplement static policies (i.e., the user’s context), (2) Designing abstractions and mechanisms that allow usable acquisition of the required information from the user, (3) Identifying the information required by the user for specifying access control policies, and (4) Designing mechanisms to account for changes in the user’s policies. Second, we discuss the need for strong security guarantees in emerging commodity platforms, such as smart homes (e.g., Samsung SmartThings). While the prevalence of user-directed information flows in these systems exposes familiar security problems, their unique functional requirements and limitations present new challenges in integrating information flow control. We discuss these challenges, along with potential solutions that direct future research.

Finally, we note that this dissertation focuses on data secrecy guarantees for application-specific user data. Within this scope, we try to provide policy and enforcement semantics to prevent unauthorized export of secret data from the device. While integrity may also be provided using information flow control [Bib77], we do not explore it in the context of application-specific data. We also do not attempt to address attacks outside the device, i.e,
on network traffic, and consider such attacks outside the scope of this dissertation.

1.2 Contributions

In this dissertation, we make the following contributions:

- **We design an extensible architecture to integrate diverse security enhancements for Android.** Modern operating systems not only expose traditional system resources to applications, but also mediate access to private user data (e.g., contacts, location) and application-specific resources. The coarse-grained nature of this mediation exposes new security risks for user data, which has prompted the development of numerous security enhancements, especially for Android. We study the semantics of the authorization hooks needed by these security enhancements, and demonstrate that enhancements solving disparate security challenges may use a set of common authorization hooks. With this intuition, we design the Android Security Modules (ASM) framework [Heu14] (described in Chapter 4), which allows security enhancements to be deployed as modules, without modifying the OS. ASM provides authorization hooks for mediating common protected operations including accesses to kernel objects (opening files, sockets), special API exposed by Android’s middleware (e.g., getting the Location, IMEI), as well as communication between application components (e.g., start activity, bind service). More importantly, ASM addresses the special challenges of modern commodity OSes, such as providing ways to modify data sent to applications, as well as allowing third-party applications to extend ASM’s interface to protect their own resources. This dissertation explores a solution for one such new security challenge, i.e., the lack of data secrecy for application-specific user data.

- **We propose a policy framework for protecting application-specific data from accidental disclosure.** On modern OSes, users often chain together applications via user-directed sharing of data, to perform complex tasks. We propose the Aquifer DIFC policy framework [NE13] (described in Chapter 5) for allowing applications to protect their privacy-sensitive data from accidental data disclosure, resulting from incorrect sharing of data by the user or by the applications involved in user-directed
tasks. In Aquifer, application developers specify the secrecy policies to protect their own application-specific data objects. Aquifer prevents unintended disclosure by enforcing the DIFC policy at the network interface, while allowing seamless sharing among applications on the device. We provide a proof-of-concept implementation of Aquifer, and integrate into Android. In doing so, we demonstrate how Aquifer can be practically realized within an existing platform. We provide three case studies by modifying popular open source applications to demonstrate the effectiveness of Aquifer’s policies in protecting data against accidental disclosure.

- We design and implement IFC enforcement that is secure as well as practical. IFC has seen limited practical adoption due to the security and practicality limitations of its enforcement on commodity operating systems. We identify the challenges of enforcing DIFC on Android, and observe that while some challenges are specific to Android, some apply to the trade-off between security and privacy in DIFC enforcement in general. We develop the primitive of “lazy polyinstantiation” for context-sensitive DIFC label propagation, and discover that adding context-sensitivity to label propagation (or floating labels) resolves the fundamental trade-off of soundness (i.e., security) vs precision (i.e., practicality) in DIFC enforcement. We design the Weir DIFC system [Nad16] (described in Chapter 6) by modifying Android, and leverage Android indirect inter-component communication to implement lazy polyinstantiation. Our initial performance evaluation of Weir demonstrates less than 4ms overhead for starting components. We demonstrate Weir’s utility with a case study using the K-9 Mail application.

- We design an approach for policy specification for application-specific user data. Policy specification for personal user data is a hard problem because it depends on many factors that cannot be predetermined by system developers. Simultaneously, systems are increasingly relying on users to make security decisions. Systems that provide strong data security guarantees for user data (e.g., Aquifer, Weir) are impractical to deploy without a method for policy specification from the user’s perspective. We introduce the Policy by Example (PyBE) paradigm (in submission, described in Chapter 7) for predicting user-specific security policies for abstract user data. PyBE takes labeled data-use scenarios (i.e., the examples) from the user, and predicts labels
(i.e., policy decisions) for new scenarios, thereby making policy specification tractable. Our prediction algorithm uses the k nearest neighbor approach, for which we develop a novel distance metric for policy examples. Our framework is cognizant of the fact that users may make mistakes in their initial examples, and uses an interactive approach to assist users in locating incorrect policy decisions in their examples. A feasibility study with expert users demonstrates 76% prediction accuracy on average, and better performance than a baseline as well as a naive approach. Finally, our interactive error correction approach finds five times as many errors as a manual review by participants.

Remark: A preliminary effort on the system described in Chapter 5 was performed as a part of the author's Masters thesis [Nad12] requirements. This effort has been expanded in Chapter 5 with a thorough evaluation and case study. Further, the ASM framework described in Chapter 4 was a joint effort with Stephan Heuser, William Enck and Ahmad-Reza Sadeghi. The author and Stephan Heuser were credited as co-first authors, and contributed equally to the implementation of the ASM framework. The author’s specific contributions include the design and implementation of a majority of the middleware authorization hooks, authorization hook placement in third-party applications, the data modification abstraction, and the implementation of the two modules for demonstrative purposes.

1.3 Dissertation Outline

The goal of this dissertation is to investigate practical solutions for user data secrecy on modern commodity operating system. We now provide a brief outline of the rest of this dissertation.

In Chapter 2, we describe the unique abstractions in modern OSes, which require and enable data secrecy policies and enforcement. Following this general discussion, we focus on specific aspects of Android's general architecture, application model and security framework that are relevant with respect to the contributions of this dissertation. We end the chapter by providing a primer on the foundational concepts of information flow control policy and enforcement.
Following the background, we discuss related work in Chapter 3. Specifically, Chapter 3 provides insights into the related work in three primary domains: OS security, modern OS security, and policy specification. Along with general takeaways from prior work, the chapter also describes new threats to user data on modern commodity operating systems.

Chapter 4 describes the unique access control semantics of modern operating systems required to address the threats discussed in Chapter 3. It then describes the ASM framework [Heu14], a programmable and extensible access control enhancement for Android.

Chapter 5 describes Aquifer [NE13], our policy framework that demonstrates the utility of DIFC policy for common data use cases on Android, and also explores novel policy abstractions for Android applications.

While Aquifer provides policy primitives tailored to the user-directed sharing in modern commodity platforms, it relaxes security guarantees for practicality. Chapter 6 describes Weir [Nad16], our DIFC system for Android that demonstrates how context sensitivity can enable DIFC enforcement that is secure as well as practical.

Chapter 7 describes the Policy by Example (PyBE) paradigm for user-specific security policy specification. We demonstrate the feasibility of our approach with a user study, and its effectiveness over purely manual efforts and naive approaches.

Finally, Chapter 8 discusses directions for future work. We reiterate the need to account for user-directed data sharing when designing the security architectures of emerging platforms, and propose future directions in the areas of information flow control and security policy specification.
Modern commodity operating platforms are primarily designed with the goal of supporting third party applications. Applications offer a variety of services not directly supported by the OS, and may even enhance or replace core OS functionality. For the purpose of this dissertation, we consider the security of user data on modern operating system architectures such as Android, iOS and Windows Phone. Specifically, we focus on providing practical and secure data secrecy guarantees to third party applications, in order to protect application-specific user data.

This chapter provides background on modern operating systems, and the fundamental concepts and terminology of information flow control (IFC) required to understand the remaining chapters. We begin by describing the general characteristics of modern operating systems, which is followed by a brief description of the secrecy challenges for user data. We then provide a primer on the Android OS, with a specific focus on its key abstractions that pose challenges or aid in practical IFC enforcement, and its access control framework. Finally, we describe the general aspects of information flow control that we use or enhance.
in later chapters.

## 2.1 Modern Operating Systems

This dissertation focuses on the characteristics of modern operating systems that are relevant for the security of user data. We identify two general characteristics that are present in all three popular modern commodity OSes, namely Android, iOS and Windows Phone, i.e., (1) applications designated as first-class security principals and (2) user-directed data sharing among applications. We discuss these characteristics, followed by a description of the new secrecy challenges for user data that are exposed as a result of these characteristics.

### 2.1.1 Applications as Security Principals

Modern operating systems take the suggestion of decades of security research [Wic90; Ioa02; ST05; Enc08] and treat applications as unique security principals. That is, an application executed by the user does not gain the full authority of the user, unlike access control in traditional operating systems (e.g., Linux). Every application is a security principal with its own set of privileges, which are a subset of the user's privileges.

As a result, an application run by the user (e.g., an antivirus) does not have direct access to data imported or created via another application (e.g., the user's phone book maintained by a Contacts application). Access must be explicitly granted by the user via user-directed sharing of information, or by the application that owns the data. Being designated as security principals allows applications to protect their data from unauthorized access by other applications, and provides applications with ownership and control over the data they bring to the device. The data that is protected may not just be limited to traditional private data associated with the user or the device (e.g., the IMEI number, the user's location, the user's contacts), but may also include application-specific user data such as notes, document scans and email attachments.

Treating applications as security principals is a significant development from the security perspective, and provides a first line of defense against trojans. However, this model also changes how applications and the user interact.
2.1.2 User-directed Information Sharing

As described previously, applications developed for modern commodity platforms are often purpose-specific. It is typical for applications to focus on a core purpose, while relying on other applications for auxiliary functions; e.g., the camera application allows the user to take pictures, but does not implement complex photo editing features, which may instead be found in a photo editor application such as Adobe Photoshop. For complex tasks that involve more than one kind of processing, the OS allows users to link together the functionality of multiple applications. To facilitate such linking, applications (i.e., the data owners) have to share data as per the user's direction.

Consider the Android platform where applications are often designed to work together to seamlessly perform large, user-defined tasks. For example, a shopping app might: (1) invoke a barcode scanner app that uses the camera to read the UPC from an item, (2) look up that item on the Web, and then (3) use a social networking app to share the item and best deal with friends. This modularity strikes a balance between simple UNIX tools (e.g., sed, grep) and monolithic GUI applications (e.g., MS Office). Android, Windows Phone and even iOS expose abstractions to facilitate such data sharing between applications [And16a; Mic13; App13]. Android applications use intent messages addressed to action strings to help the OS find the best application for a task. Similarly, Windows Phone provides “share charms” to help users complete tasks with different applications. Finally, iOS provides limited sharing and navigation between applications using URL protocol handlers.

While data sharing among applications facilitates complex user tasks along with a seamless smartphone experience, it also exposes new challenges for user data security, as we discuss in the next section.

2.1.3 User Data Secrecy in Modern Operating Systems

While applications control the access to their data, they do not control the flow of their data once it is shared. The access control mechanisms for application data on modern OSes (e.g., Android permissions) are only effective at the first point of access; i.e., when the initial sharing takes place. The security policy regulating a data object does not apply after it is copied to another application's memory. This lack of control due to non-transitive
enforcement exposes application-specific user data to the risk of unauthorized disclosure to the network. For example, a photo of a whiteboard containing meeting notes might be inadvertently shared and uploaded to a social networking site, or a confidential email attachment might be exported to a remote adversary by a malicious document viewer. A key challenge for modern OS security is controlling the flow of data between apps and preventing accidental or malicious information disclosure.

Preventing data disclosure is not as simple as restricting the set of applications that an application with access to sensitive data can interact with (e.g., Saint [Ong09]). A trusted application receiving data might share that data with another application that has unexpected disclosure. Hence, in the collaborative application environment, we must address the accidental disclosure problem as one of information flow, by integrating information flow secrecy guarantees into modern operating systems. In fact, the lack of data secrecy is one of the fundamental problems addressed by information flow control (IFC) systems [Den76; BL73]. However, the data flows described in prior solutions are often among users running known user programs or system applications with different labels, and are often possible to predict a priori. Providing data secrecy is much more challenging and complex in modern operating systems, where a single user dynamically directs data sharing between mutually mistrustful applications at runtime, and where the resulting flows may be unpredictable from a centralized perspective.

2.2 Android

Android is an operating system for mobile devices, first released in 2008. As of late 2015, Android has more than 82% of the market share, the largest of all smartphone platforms [IDC16]. Aside from being the most popular smartphone platform, Android is also the most advanced of all modern OSes in terms of its extensibility and application interoperability. Android allows applications to replace core OS functions (e.g., file explorers, camera applications, dialers) and hence applications often require broad access to private user data. Further, Android’s application model encourages interoperability and data sharing with other applications, in order to deliver a seamless experience to the user while executing tasks that require more than one application. These factors, along with the availability of source code
for experimentation, make Android a suitable candidate for research into the security of the emerging modern operating systems.

This section focuses the unique abstractions in Android that enable advanced data security mechanisms such as the ones described in later chapters. We start by providing a brief overview of Android's system architecture. We then describe Android's application model, which requires applications to be divided into modules with fixed functions and abilities. The modularization of applications, and the control over the different functional aspects of an application, allows fine-grained access control on Android. We then discuss Android's inter-component communication abstractions that allow the OS to mediate interactions among components, following which we describe the life-cycles of Android's component types. Finally, we discuss the security architecture of Android, with a specific focus on the policy and enforcement for protecting user and system data.

2.2.1 Architectural Overview

Figure 2.1 shows Android's system architecture. Android runs a Linux kernel, but defines its own application runtime environment. The kernel enforces isolation between applications and access control for some resources for which it has context (e.g., Internet).

Android applications use the middleware API to request hardware and software resources that the kernel does not regulate (e.g., Location coordinates). As a result, some policy decisions for high-level resources are taken by the reference monitor implemented

---

1The figure only shows layers that are relevant to this dissertation, while other layers (e.g., the hardware abstraction layer (HAL)) have been omitted for simplicity
in trusted system services that operate in the middleware layer. For example, the Activity Manager service regulates interactions between application components, while the Package Manager service controls aspects of an applications installation and removal.

The Java-based middleware API forces developers to design their applications within a component framework. Android defines four component types: activity, service, content provider, and broadcast receiver. We now describe Android's components, followed by the inter-component communication in Android.

### 2.2.2 Application Model

The Android application model consists of four components, namely activities, services, content providers, and broadcast receivers. Activity components define the application's user interface (UI). Each UI screen is defined by a different activity component. The other components types run in the background and are started by the Android middleware as needed. These component types provide daemon-like functionality. Service components are general purpose daemons; content provider components provide a uniform interface to application data, and broadcast receiver components listen for asynchronous broadcast messages from the system or other applications.

Android uses the abstraction of a task (i.e., a collection of activities) to denote complex user task involving more than one activity component. Activities are foreground components that can have multiple simultaneous instances in different user tasks [Andb]. Activity instances are placed in a “stopped” state when not in the foreground, but remain in the system's memory, to be killed only in low memory conditions. On the contrary, broadcast receivers have very short lifetimes, and are aggressively destroyed as soon as they finish processing the broadcast. Service and content provider components can have only one active instance at a time, that can continue to operate in the background unless they are deliberately stopped or exit by themselves.

Additionally, Android provides the “android:launchMode” manifest attribute for developers to manage activity instances, as follows: The default standard launch mode launches a new instance of the activity per call, in the caller's user task. An instance of an activity launched as singleTop is resumed for new calls, if it already exists at the top of the UI stack of activities at the time of the call. An activity launched as singleTask or singleInstance is
allocated a separate task of its own. Every call to such an activity resumes its instance in the said task. The only difference between the two modes is that the singleInstance activity can be the only activity in its own task, whereas other activities can be placed in a singleTask activity’s task.

2.2.3 Inter-Component Communication

Android’s binder framework (backed by the binder kernel driver) provides process control and IPC between components. Applications generally do not interact with binder directly. Instead, they use intent messages, which start activity and service components, and send messages to broadcast receiver components. The key attribute of intent messages is their ability to be sent to implicit addresses. For this, Android uses action strings, such as ACTION_VIEW and ACTION_SEND. Applications define intent filters to register to receive messages addressed to specific action strings. The Android framework then automatically determines potential intent message destinations (i.e. resolves the intent), presenting the user with a list of targets if a single destination must be chosen from a set.

From the IPC mechanism perspective, inter-component communication on Android can be divided into two categories: 1) indirect and 2) direct. Indirect communication involves an asynchronous call from one component to another, through the Activity Manager service (e.g., startActivity, bindService). Direct communication involves a synchronous Binder remote procedure call (RPC) using the “Binder object” of the callee component to call a remote method defined in that component. Note that while direct communication does not require the Activity Manager, its setup involves an asynchronous call to the Activity Manager to retrieve the callee’s Binder object.² For example, the first operation executed on a content provider (e.g., query, update, delete, insert) by a caller is routed through the Activity Manager service, which retrieves the Binder object of the target content provider, and loads it into the memory of the caller. Future calls are routed directly to the target content provider. Similarly, “bindService” sets up a Binder connection to a target service via the Activity Manager, to be later used for Binder RPC.

²Binder objects for system services are stored with the Service Manager.
2.2.4 Android's Security Architecture

Android is a multi-user Linux system, where applications are installed as Linux users with their own unique UIDs. The UID (and GID for some resources such as the Internet) of an application are used during enforcement. That is, applications are treated as first class security principals in their own right, and granted access to only a subset of device’s data and functions available based on their identity and group membership. Further, applications are granted their own private storage, and access to application component instances as well as data is regulated using application UIDs.

Android’s permissions (i.e., text strings that represent resources) are used to control access to system and user resources. Applications request for permissions to protected resources statically, in their application manifest files. Permissions protect access to protected resources (e.g., Internet, bluetooth), user data (e.g., Contacts, Call Logs), and access to an application’s own components as well. That is, applications can define their own permissions, or use system-defined permissions, to restrict access to specific components.

Before Android version 6 (i.e., Android Marshmallow), the user had to agree to all the permissions requested by an application before it could be installed. This all-or-nothing deployment model made it difficult for users to install applications with only a subset of the permissions requested, resulting in the problem of over-privileged applications [Enc11; Fel11a; Au11; Ber11; Nad14]. To mitigate the drawbacks of the install-time permission model, Google recently introduced runtime permissions in Android 6.0. The new model allows users to deny some or all of the permissions demanded by applications through runtime prompts as well as application settings.

Once a permission is allocated to an application (using the install-time or runtime approach), it is associated with the application package, and hence the assigned UID or GID for enforcement. To protect sensitive system API, Android’s system services (i.e., middleware) retrieve the UID (supplied in the Binder call), and query the Package Manager service’s checkPermission() function to resolve access by the specific UID to the required permission for that API. Third party applications can also use the same method to protect their Binder-accessible API. Third party applications can also set their components to only

\[^3\]A single UID may be “shared” by more than one application with the same signature, and hence allow applications to combine privileges
be internally accessible, by setting the exported manifest attribute of the component to false. To grant temporary access to subsets of data backed by content providers, applications can grant “URI permissions” for specific content provider URIs. However, not all functions are protected using checkPermission(). Some low level resources are protected using Linux GIDs (i.e., through application membership to resource-specific groups such as sdcard_rw for SD card read-write access). The GID-based access is enforced by the kernel.

A primary drawback of permission enforcement is that permissions are not transitively enforced. That is, a permission is only enforced at the first point of access (e.g., acquiring a URI-permission protected data object from an application). Once the data object is copied in the memory of the adversary, permissions do not prevent unauthorized uses of the data, such as exporting it to an untrusted remote server. As a result, permissions cannot be used to ensure data secrecy.

Android relies on Linux Discretionary Access control (DAC) to enforce its UID-based isolation. To overcome the shortcomings of DAC [Los98], Android enforces type-based Mandatory Access Control through SEAndroid [SC13]. To elaborate, SEAndroid enforces least privilege on daemons that execute with root and system privileges, and to control the damage in case such daemons are compromised and fall prey to confused deputy attacks. While SEAndroid may be configured to provide transitive enforcement for system resources, its policy specification is centralized, limiting its ability to accurately express the secrecy requirements of dynamic application-specific user data (e.g., Emails, notes) that are only known to the user or the application.

2.3 DIFC Fundamentals

Information Flow Control (IFC) facilitates the definition and enforcement of the allowable data flows in the system. In an IFC system, the security contexts of subjects (e.g., processes) and objects (e.g., files) are composed of predefined security classes that denote secrecy levels (e.g., top-secret, secret, confidential). The secrecy policy determines the data flow (i.e., ordering) between any two classes based on a partially ordered finite lattice [Den76]. Labels may also be joined to form a higher label in the lattice. In such a lattice, data can only flow up, i.e., to a higher security class. Data flows violating the lattice require explicit authorization
(i.e., declassification) by the policy administrator. Thus, IFC ensures that secret user data only leaves the device through authorized channels, preventing the unauthorized data disclosure described in the motivating example.

Traditionally, IFC systems (e.g., MLS [BL73]) use a central definition of security labels, which does not meet the needs of applications with varying secrecy requirements for their data. Myers and Liskov [ML97] defined a decentralized label model (DLM) that allowed applications to extend the IFC lattice with their own labels. Many such decentralized information flow control (DIFC) policy models and systems (e.g., Asbestos [Van07], HiStar [Zel06], Flume [Kro07; KT09], Laminar [Roy09], and Fabric [Liu09a; Ard12]) have since been proposed, that extend the enforcement of IFC guarantees to include unknown subjects and objects, such as app-defined components and data. We now describe the fundamental aspects of DIFC, which directly motivate the challenges to making DIFC practical raised in this dissertation (specifically Chapter 6).

2.3.0.1 Label Definition

In a DIFC system, security principals create labels (i.e., security classes) for their own secret data. The resultant lattice is infinite, and dynamically adapts to the secrecy policy of each data owning security principal. On Android, decentralized label definition would allow apps to articulate trust on a fine granularity by creating labels for their data, and controlling assignment of these labels to subjects and objects, eventually controlling the flow of their data. Note that while DIFC can also provide integrity guarantees, we describe DIFC with respect to data secrecy as it is the focus of this dissertation.

2.3.0.2 Label Changes and Floating Labels

The finality of subject and object label assignment is called tranquility, an important property of mandatory protection systems. Tranquility constraints have to be relaxed for a decentralized IFC policy. Data owners may then control the ability of subjects to explicitly raise or lower their labels, through the assignment of privileges (i.e., capabilities) related to the security classes. The mechanism ensures that label changes using a data owner's security class are safe, i.e., permitted by the data owner.

While explicit label changes offer some flexibility over immutable labels, they are not
practical for general-purpose subjects that may be invoked from various secrecy contexts. To support such cases, floating label DIFC systems (e.g., Asbestos [Van07]) allow seamless data flows through implicit label propagation. That is, the caller's and the callee's labels are joined, and the resultant label is set as the callee's label. Floating labels make DIFC compatible with unmodified applications, and are required on Android, where inter-application communication is often user-directed and unpredictable.

### 2.3.0.3 Declassification

DIFC enforcement only allows data flows permitted by the policy lattice, and any exceptions require authorization (i.e., declassification) by the data owner. Thus, declassification provides the data owners with control over the export of their data. Data owners may choose to explicitly declassify data whenever required, or delegate other security principals to declassify on their behalf. Further, as DIFC enforcement is limited to the device, data has to be declassified before it is exported to the network. In an Internet-driven environment where security principals may always be connected to the network, explicit declassification is impractical due to its frequency. Another alternative is delegating the declassification capability to security principals that may potentially connect to the Internet, which would bloat the application's TCB. Practical and secure declassification for network export is thus a challenging aspect of DIFC design.

### 2.3.0.4 System Integration

One of the first steps while integrating data secrecy into an existing OS is the selection of the subject for data flow tracking. Fine-grained dynamic taint tracking (e.g., TaintDroid [Enc10]) labels programming language objects to provide precision, but does not protect against implicit flows. OS-based DIFC approaches [Zel06; Kro07] adopt the better mediated OS process granularity, but incur high false positives; i.e., functions sharing the process with unrelated functions that read secret data may be over-restricted. While secure process-level labeling is desired, practical DIFC enforcement must minimize its impact on functionality.
The objective of this dissertation is to provide strong security guarantees for user data on modern commodity operating systems, through practical and secure policy and enforcement primitives. As a result, the contributions of later chapters draw upon the lessons and insights learned from prior research in the fields of operating systems security, Android security and policy specification. This chapter describes relevant prior work in these areas.

### 3.1 Operating Systems Security

As described by Lampson in 1974, the original motivation for integrating protection mechanisms into computer systems was to prevent one user’s malice or faults from harming other users [Lam71]. It was identified that the “harm” itself can be inflicted in three ways: (1) by copying another user’s confidential data, (2) by destroying or corrupting another user’s data, and (3) by degrading or making services unavailable for another user. The
potential for harm in these previously described ways is what motivates the fundamental
security goals of secrecy, integrity, and availability. As Lampson argued, the threats that
affect users also similarly affect “programs” and other runtime abstractions; hence, it is
necessary to take an abstract approach to treat these security concerns, rather than a few ad
hoc mechanisms. The domain of operating systems security performs this crucial function.
Given that operating systems form the backbone of modern computing, OS security can be
said to be one of the most essential areas of computer science.

In an operating system, the execution context of a program is known as a process. Like
many of the core concepts of OS security, the process abstraction was first developed in
Multics [Sal74], and provided separation between the runtime contexts of different user
programs. Each process has a protection domain, which defines the degree of access a
process has to system and user resources. An access control matrix may be used to represent
all the protection domains in the system [Lam71]. The rows of the access control matrix
are labeled by a set of subjects (e.g., processes, users) or subject domains, and its columns
are labeled by a set of objects (e.g., processes, files). In simple terms, a subject is an entity
that may be associated with an execution context of a program, whereas an object is the
generalization of all things that may hold information. The cells of the matrix are the actions
that the subjects can perform on the objects (e.g., read, write, execute). We refer the reader
to Jaeger’s far-reaching book [Jae08] for elaborate explanations of these concepts, as well as
for discussions on orthogonal topics in OS security not covered by this section.

In practice, an access control matrix is a sparse data structure. Therefore, alternate
representations are used in place of the matrix, namely access control lists (ACL) and
capability lists (CL). An access control list is used to store the access rights related to an
object (i.e., the column in the matrix) with the object. Alternately, a capability list (CL)
stores the access rights of a subject (i.e., the row in the matrix) with the subject. Both
alternatives have their limitations; e.g., with ACLs, revoking multiple subjects’ access to a
specific object is simple, although revoking the same subject’s access to multiple objects
may be prohibitive. The converse is true for CLs.
3.1.1 Discretionary and Mandatory Access Control

The access control matrix represents the bounds of protection, i.e., the protection state. However, it alone is not sufficient to achieve the security goals of secrecy and integrity, which depend on the protection system. Protection systems are of two types, discretionary or mandatory, based on how they are managed. A protection system where individual subjects are able to modify the protection state is known as a discretionary access control (DAC) system. In such a system, untrusted user processes can modify the protection state of objects owned or created by the user. That is, the protection state is at the discretion of users and the untrusted programs they execute, and may result in violation of the security goals of secrecy and integrity.

Owing to the discretionary management of the protection state, the safety problem was found to be undecidable for DAC systems [Har76]. That is, it may not be possible to guarantee that all possible future protection states derivable from the current protection state of a DAC system may be safe; i.e., may never allow accesses that violate secrecy or integrity goals. A mandatory access control (MAC) system, on the contrary, has a mandatory protection state, in which the access control permissions on objects are managed by an authorized administrator. Additionally, MAC systems separate the execution context of a process (which is discretionary) from its security context, which is predetermined by the administrator. As a result, the safety problem is decidable for a MAC system.

It is important to note that in commodity operating systems such as Linux, the use of MAC is limited to protecting system resources, such as system daemons and critical system data. MAC protection for third-party application resources is scarce, and for a valid reason: system policy administrators cannot reason about the secrecy and integrity requirements of user data generated via third party applications. As a result, DAC enforcement is often used in conjunction with MAC. For example, Android uses SEAndroid to sandbox system daemons, but uses Linux user-based DAC enforcement to isolate private application data.

3.1.2 Information Flow Control (IFC)

Mandatory protection systems have been historically described in terms of subject and object labels, and well-defined flows between the labels. A label defines the security context
of the subject or object. Jaeger defines a mandatory protection system as composed of (1) a mandatory protection state, which is defined in terms of subject and object labels, and the privileges subject labels have over object labels, (2) a labeling state, which maps subjects and objects to labels, and (3) a transition state, which describes legal transitions between labels, i.e., rules for relabeling subjects and objects [Jae08]. Using this definition, a MAC policy may control the flow of information in the system, by labeling sensitive and public data objects, and regulating read/write accesses and label transitions between sensitive and non-sensitive labels. That is, MAC systems can be designed to provide provable information flow guarantees of data secrecy, integrity, or both.

In 1973, Bell and LaPadula formalized the US Department of Defense's Multi-level Security (MLS) policy for providing access control for military computers and applications [BL73]. The resultant Bell-LaPadula model is a mandatory protection system whose labels and labeling state are predetermined by a trusted administrator. The model provides data secrecy, guaranteed by two properties in particular: the simple security property (i.e., “no read up”), which states that a subject must not read data from an object at a higher security level, and the *-property (i.e., “no write down”), which states that a subject must not write to an object at a lower security level. Both these properties ensure information flow from lower to higher security levels. The model also defines a tranquility principle, which requires that the label of an object may not be changed during the normal operation of the system. As a result, dynamically upgrading an object for the *-property to hold may not be possible under the model. This aspect of the model limits its practicality in non-military systems, as we discuss further in this chapter.

Just as the Bell-LaPadula model formalizes secrecy, the Biba model [Bib77] represents its dual and defines the access control rules for guaranteeing data integrity. More precisely, it defines the simple integrity property (i.e., “no read down”) and the *-integrity property (i.e., “no write up”), with the notion of integrity being that a process must behave as it is expected, if it does not rely on low-integrity inputs. It is important to note that secrecy and integrity requirements are duals of one another. Achieving both simultaneously in one system is possible, but only when the system uses two disjoint sets of labels, one for each guarantee. The Flume model [Kro07], described later in this chapter, is a good example if such a system that provides both secrecy and integrity.

As described previously, information flow control (IFC) model such as the Bell-LaPadula
model assign subjects and objects with labels, or security classes, and define the security policy in terms of the allowable flows between the labels. Denning defined a lattice representation for IFC policies, where the policy would be represented in terms of a lattice of labels [Den76]. In Denning's lattice model, the allowable information flows between two classes are represented by a binary can-flow (→) relation, which governs the order of the two classes in the IFC lattice. Further, a binary join operator (⊕) allows distinct classes to be joined to result in another class in the lattice. As a result, the system's IFC policy can be entirely represented in the IFC lattice.

The IFC lattice is a verifiable representation of the system's IFC policy, but only for previously known subjects and objects. That is, the IFC lattice cannot incorporate the security policy for new types of secret objects created by applications in the system, which may not be known to the administrator beforehand. For example, a Web server may want to provide secrecy guarantees for private data generated by its clients, by assigning a different security class to each client's private data objects. Such requirements are also common in modern commodity operating systems, where third party applications often have secrecy or integrity requirements that may not known to the administrator a priori. Decentralized information flow control (DIFC) supports such requirements, by allowing the IFC lattice to be extended by data owners other than the administrator, in a way that only makes the existing IFC policy more restrictive. While Section 2.3 provides general background information on DIFC, we now describe specific prior DIFC systems and policy models.

### 3.1.3 Decentralized Information Flow Control (DIFC)

Decentralized information flow control (DIFC) is a generalization of the Decentralized Labeling Model (DLM) created by Myers and Liskov [ML97]. DLM allows users or applications (or more generally data owners) to create new security classes for their own secrecy requirements, and extends the IFC lattice with these classes. A key aspect of DLM is that it does not rely on a central, trusted service to declassify data, which is infeasible in a decentralized environment with mutually distrustful security principals. Instead, each data owner controls the ability declassify its own data. DLM allows owners to administer the IFC policy (i.e., the can-flow relation), but only for the security classes they create. That is, a data owner may specify readers for data labeled with their own security classes, but
may not specify readers for any other class. As a result, the DLM model, and DIFC systems in general, only make the enforcement more restrictive by further restricting information flows for application data.

Based on the approach, DIFC systems can be loosely classified into three types, namely language-based, OS-based and hybrid systems. We discuss examples and trade-offs of each approach as follows:

**Language-based DIFC**: Language-based DIFC systems (e.g., JFlow [Mye99], Jif [ML00], SIF [Cho07]) define information flow policies in terms of flows between programming language objects. While the techniques employed by such approaches may be similar (e.g., static type checking), the end objectives are diverse. For instance, JFlow [Mye99] (and later Jif [ML00]) is an extension of Java that employs static checking of information flow constraints using the decentralized labeling model, and treats information flow checks as an extended form of type-checking. The objective of Jif and JFlow is to ensure that a program follows the information flow constraints set by mutually mistrustful security principals represented in the program. The Fabric platform [Liu09a] provides similar guarantees for confidential user objects in a distributed setting, by extending the Jif language to create the Fabric language for distributed computing. Further, SIF provides IFC support for servlets [Cho07].

Language-based DIFC has its advantages, both in terms of precision and security. Language-based systems can reason about flows at the level of a programming language object, providing information flow tracking that is more precise than using coarser abstractions such as OS processes. Additionally, compile-time checking is resilient against implicit flows (i.e., implicit value assignments resulting from control flows). Finally, static information checks incur no run-time overhead, and failures of such checks do not result in data leaks.

However, there are two major drawbacks of language-based DIFC systems. First, the security guarantees of programming language do not extend to the underlying OS abstractions, and must depend on the OS for DIFC enforcement on OS objects (e.g., processes, files, sockets). Second, developers must annotate and reason about information flows in new programming languages, which harms backwards compatibility with legacy software. Moreover, new languages present new challenges for developers, although Java-based DIFC...
languages mitigate this limitation to some extent.

**OS-based DIFC:** OS-based DIFC systems such as HiStar [Zel06] and Flume [Kro07] define information flow control policies in terms of flows between OS objects (e.g., processes, files, sockets). The DIFC systems described in this dissertation (i.e., Aquifer [NE13] and Weir [Nad16]) fall in this category. OS-based DIFC can be used to regulate data flows between well-defined pieces of a single program, as well as among multiple programs.

Research in OS-based DIFC systems has produced entirely new operating systems with novel abstractions, or has retrofitted the treatment of existing OS objects with IFC checks. Novel DIFC operating systems such as Asbestos [Efs05; Van07] and HiStar [Zel06] are of the former kind. Asbestos is primarily designed to separate the data of different Web application users connected to a common Web server, in spite of untrusted, potentially vulnerable Web application code. Its abstraction of the event process allows a single server process to isolate data from multiple user-requests into different internal process states, thereby preventing the problem of label explosion described in Section 2.3. However, it uses floating label propagation, making it prone to implicit data leaks. While HiStar [Zel06] avoids implicit data leaks by making information flows explicit, it cannot offer the seamless communication of Asbestos. A major hindrance in using Asbestos and HiStar for commodity computing is that they are fairly new and complex operating systems, and expecting application developers to redesign legacy applications to simply execute on these new systems is impractical.

Flume [Kro07], however, adapts standard UNIX abstractions for enforcing information flow control policies, making only minimal changes to communication primitives. Hence, while developers would need to make minimal modifications to applications for Flume, a complete redesign may not be required. However, similar to HiStar, Flume also requires explicit declaration of label changes, making it incompatible with ad hoc information flows. Recall that for an IFC system to provide both secrecy and integrity, it would need to use two sets of labels; otherwise, each subject’s access would be limited to only its own security level. A key aspect of Flume is that it uses two disjoint sets of labels, one for secrecy and another for integrity, enabling it to support both.

As may be evident from the previous discussions, the label change semantics of OS-based DIFC systems face the trade-off between security and compatibility. That is, implicit
label changes such as in Asbestos are prone to implicit data leaks, but offer seamless communication. On the contrary, explicit label changes or flow declarations such as in Flume or HiStar are secure, but possible only when all the flows can be predetermined. Due to user-directed sharing in modern operating systems, it is not feasible to define all flows a priori, leaving implicit label propagation as the only alternative. Chapter 6 explores a modification to the traditional implicit label propagation logic that may be resilient to data leaks. It is important to note that language-based systems do not suffer from the same trade-off, as implicit data leaks are easier to address with static checking. In fact, a simple language-based solution to this problem has been proposed by Fenton [Fen73] and Gat and Sal [GS76], that of restoring the value of the affected variable to its value before the condition, after the completion of the condition’s execution.

Although OS-based approaches are less precise, their guarantees extend to the entire operating system, i.e., from the application to the kernel, and to the network and file system interfaces. With moderate effort by the developer, OS-based systems can also secure intra-application resources (e.g., trusted components), although the precision does not extend to the language-level.

However, the lack of precision may give rise to false negatives. For example, if an application process executes multiple functional components, only one of which accesses secret information, the entire process would be labeled for that secret access by a process-granularity DIFC system (e.g., Flume). Some systems attempt to reduce false negatives within the process by taking a hybrid approach, as we discuss next.

Hybrid (OS and Language-level) DIFC: Laminar [Roy09; Por14] provides both language-level as well as the OS-level enforcement. For the language-level enforcement, it modifies the Java virtual machine (JVM). However, while the process-level enforcement applies to programs modified to use Laminar and Laminar-unaware programs alike, the fine-grained language-level enforcement is only available to programs that explicitly modify their code (i.e., akin to language-level DIFC systems). This design choice is infeasible for a practical DIFC system that may need to be backwards compatible with unmodified applications. The approach proposed in Chapter 6 provides precise information flow tracking along with backwards compatibility with applications.
3.1.4 DIFC on Android

This section describes prior DIFC systems for Android (i.e., Jia et al. [Jia13] and Maxoid [XW15]. Note that Aquifer [NE13], our contribution in Chapter 5 preceded Maxoid, and was one of the first DIFC systems for Android along with the system proposed by Jia et al. This section provides a brief overview of these prior approaches; Chapter 6 provides a detailed explanation of the enforcement challenges of DIFC on Android, the lessons learned from prior work.

**Jia et al.** In their ESORICS’13 paper [Jia13] (supported by the technical report by Aljuraidan et al. [Alj12]), Jia et al. port the Flume DIFC Logic [Kro07] to Android. The paper uses floating labels, to allow multi-purpose applications (e.g., editors and viewers) to be instantiated with the caller’s labels. A key contribution of the proposal is its non-interference proof.

To prevent false positives due to process-level labeling, the proposal treats Android applications and components as single-instance, and blocks on every new call until the component or application voluntarily exits and the process dies. However, this approach is contradictory to the traditional lifecycles of Android components and applications, which we described in Chapter 2. Hence, the approach is not backwards compatible with existing applications. Additionally, the proposal does not account for label explosion via storage writes, which is a major concern for DIFC enforcement.

**Maxoid:** Maxoid [XW15] is a recent system that attempts to solve the problem of label explosion via storage using a file system polyinstantiation approach, first described in Solaris containers [LS05]. Maxoid groups application instances into chains of initiators and delegates, similar to Aquifer’s workflows (described in Chapter 5), and enforces the DIFC policy defined by the initiators.

Although Maxoid and Jia et al. both use floating labels, Maxoid offers an improvement in terms of label propagation to storage, by ensuring that delegate instances started by different initiators (i.e., labeled differently) receive separate storage through the use of a copy-on-write file system. As a result, label explosion via storage is contained. However, Maxoid does not provide similar support for process memory, and instead has to kill the process for each new call. The approach is also incompatible with label propagation via background components, which is one of our key contributions in Chapter 6.
3.2 Modern OS Security

This dissertation is not the first to consider the challenges of modern operating systems. The ServiceOS project at Microsoft Research, which includes MashupOS [Wan07] and Gazelle [Wan09], considers similar problems, but focuses on Web browsers. Also under this umbrella project is Access Control Gadgets (ACG) [Roe12], which uses trusted UI widgets to infer user intentions when accessing sensors (e.g., camera, microphone). ACGs are a generalization of the much earlier concept of a “powerbox,” which is a trusted dialog box originally used by CapDesk [Cap] and DarpaBrowser [Dar] to grant a process access to a file based on the user’s natural file selection process. The research presented in this dissertation applies the broad insights presented in these prior works inspire the data secrecy approach for modern commodity operating systems described in this dissertation.

It is well-known that on modern operating systems, third-party applications assist the user in creating, managing and processing information. For performing these functions, applications must be able to add user data to the system, and access existing user data for processing, on the user’s behalf. Both these requirements motivate and necessitate user data security on modern commodity platforms such as Android. In this section, we first motivate the need for user data security on modern commodity OSes by describing threats to application-specific (e.g., notes, email attachments) and generic private user information (e.g., location coordinates, IMEI). We then describe the defenses discussed in prior work, with a specific focus on work that directly or indirectly aims to ensure the confidentiality of user data. Our discussion is centered around the Android platform, which is the target platform for this dissertation.

3.2.1 Threats to User Data

Chapter 2 describes the security architecture of Android, i.e., the defense mechanisms deployed to protect user and resources, and the factors motivating them. In this section, we take an alternate route, by describing the threats shown to be feasible on Android in spite of the existing defenses. Note that modern computing devices have a vast attack surface, and we do not aim to provide a comprehensive summary of possible attacks or threats. Instead, we focus on the threats that may directly or indirectly endanger user data.
We first provide a quick recap of Android's permission model, followed by a description of the three application-level threats to user data that the permission model cannot address.

**Revisiting Android's Permission Model:** Android treats applications as security principals, and more precisely, as “users” in a Linux UID-based discretionary access control (DAC) system. At any point of time, an application's privileges are bound by the permissions it has been granted. The Android platform places no restrictions on the permissions requested by third party applications [Enc09b]. The decision of granting the requested permission(s) rests with the user. Prior to Android Marshmallow (i.e., version 6.0), Android's install-time permission model only allowed users to accept all permissions at install-time, or reject the installation of the application. Android version 6.0 (and later) supports runtime permissions, allowing the user to dynamically reduce application permissions.

### 3.2.1.1 Application-level Threats

As previously discussed in Chapter 2, a fundamental drawback of the permission model is its lack of *transitivity*. That is, permissions can only be used to enforce checks on the first access, but not on the subsequent flow of information. As a result, an application that can access information can also share it with other applications or export it to the network without the user's authorization. We now discuss other threats to user data from similar drawbacks of the permission model.

#### 1. Over-privileged Applications:

An over-privileged application is one that possesses more permissions than required for performing its advertised functions. Numerous studies have demonstrated the prevalence of such over-privileged applications on devices running Android Lollipop (i.e., version 5) or prior [Enc11; Fell1a; Au11; Ber11; Nad14]. Over-privileged applications violate the principle of least privilege, and may compromise private user information (e.g., a flashlight app that tracks the user's location), or may be inadvertently exploited as confused deputies by malicious adversaries. Runtime permissions in Android 6.0 have improved this status quo for a minority of Android users (above 26% [And16b]), by providing users with fine-grained control over application permissions. However, prior studies have demonstrated that users generally do not understand permissions and their implications [Fel12; Kel12]. Thus, delegating all security policy decisions to the user may not be an effective approach for complex systems, which motivates some of the defenses.
discussed later [Roe12], and the policy specification approach presented in Chapter 7.

2. Confused Deputy and Collusion: The confused deputy problem is a classic problem associated with discretionary access control (DAC) systems, where an adversary may trick a privileged security principal into misusing its privileges for the benefit of the adversary. Felt et al. identified this problem as permission re-delegation on Android, where an application with permissions is tricked into performing a privileged operation (e.g., accessing the user's location) for a malicious application without permissions [Fel11b]. A collusion attack, on the contrary, requires active participation of multiple applications to perform a privileged task that is beyond their individual capabilities; e.g., an application with only the location access permission (but not Internet access permission) may collude with an application with the Internet access permission (but not the location permission), in order to exfiltrate the user's location. Prior work has demonstrated the feasibility and existence of such threats to the user's private information on Android [Fel11b; Lin10; Gra12a].

3. Lack of Intra-application Privilege Separation: Android applications are often driven by revenue from advertisements, and must include third party advertisement libraries in their installation packages. As Android does not provide privilege-separation between application code and third party libraries, the ad libraries often execute in the application's execution context. The implications of this are twofold: first, applications may need to request privileges solely required by the ad library, making the application code over-privileged [Pea12], and second, ad libraries may execute with the application's privileges, making them over-privileged, and exposing private user information and critical device resources to ad libraries [Ste12]. The lack of privilege separation between application and library code may allow ad libraries to surreptitiously violate the user's privacy.

3.2.1.2 Indirect Threats from OS Compromise

While some private user resources and privileges can be mediated by the kernel (e.g., files, Internet access), modern commodity OSes also offer several abstract resources that are not represented in the kernel, and require mediation by OS middleware executing in user space. For instance, Android's application component model and inter-component communication is entirely managed by the Activity Manager service. Such middleware services often execute at a higher privilege level (i.e., as the “system” user). Note that although system
services have less privileges than the “root” user, the compromise of a crucial system service may be sufficient to override many of the inter-application access control protections, and allow the adversary to gain access to private application-specific user data. While a manual analysis of such services has previously led researchers to vulnerabilities [Xin14; Che14], a more systematic security analysis of system services is necessary.

Additionally, vulnerabilities in system daemons (e.g., the vulnerability in vo1d [Per14]) or the Linux kernel (e.g., towelroot [Hot14] have also been exploited to gain root privileges. An adversary with root access to the system can override most security protections for application and user data.

3.2.2 Defense of User and Application Data

The security of mobile platforms and applications has received much attention in the last decade. Android, in particular, has been the focus of most of this research, owing to its popularity and the availability of platform source code. In this section, we aim to capture the key defenses proposed by prior work for directly or indirectly ensuring the security of user data. We describe three categories of research, namely (1) research that uses static program analysis, (2) research that uses dynamic program analysis, and (3) research that uses run-time enforcement to enhance the security architecture of the OS.

3.2.2.1 Static Program Analysis

Static analysis is focused on identifying the presence or absence of predefined properties in program code or metadata, without executing the target program. Prior work has applied static analysis to third-party Android applications as well as to parts of the operating system.

Some undesirable properties of target programs may be apparent from simple analysis on program metadata (e.g., application permissions), and may not need heavyweight reverse engineering of application code. For instance, Kirin [Enc09a] identifies potentially harmful applications using undesirable permission configurations; i.e., through predefined rules that describe which permission combinations within a single application that could be used for malicious purposes. Similarly, DroidRanger [Zho12] uses permission patterns to recognize members of known malware families. However, such analysis is extremely coarse-grained, and fraught with false positives.
Fine-grained static analysis of application code is slower than light-weight metadata analysis, but provides greater precision. To facilitate the use of existing static analysis frameworks for Java with Android applications, prior work retargets Android's DEX bytecode to Java bytecode [Enc11; Oct12]. However, Android's inter-component communication and other unique abstractions make the direct application of Java static analysis frameworks difficult. To this end, prior work has developed numerous static analysis frameworks and tools for targeting Android applications for various purposes, such as detecting private data leaks in Android applications [Fri14; Gib12; Li15; Gor15] or ad libraries [Gra12b], detecting malware [Zha14; Arp14], generating precise inter-application communication mappings [Oct13; Chi11], detecting over-privileged applications [Fel11a], identifying the presence of specific inter-component communication vulnerabilities [Lu12], performing inter-component data flow analysis [FWR14], or identifying flaws in the use of cryptographic primitives and protocols [Fah12; Ege13].

Prior work has also applied static analysis to investigate the security of the Android OS middleware [Cao15; Sha16]. EDGEMINER [Cao15] identifies implicit control flow transfers (i.e., callbacks) placed in the Android framework, which allows it to provide a more accurate view of data flows within the Android framework. While this analysis does not target the security within the framework itself, it assists in identifying information flow leaks in applications that may leverage such callbacks to evade detection. On the contrary, Kratos [Sha16] analyzes the permission checks in the Android middleware, and identifies several missing checks using the consistency of check placement as a metric. The findings of this work provide motivation for a thorough security evaluation of the Android framework, especially its critical services (e.g., the Activity Manager, Package Manager) that enforce application-level security policies. Prior work has also used static analysis to demonstrate vulnerabilities in OEM applications that may allow an adversary to side-step Android's permission enforcement [Gra12a; Wu13].

Systems that perform static analysis often fail in scenarios that require the runtime context, regardless of improvement in the static analysis technique; e.g., instance, EPICC [Oct13] resolves inter-component communication with significant improvement in precision over its predecessor ComDroid [Chi11], but cannot resolve scenarios where the runtime context decides the string arguments. More importantly, static analysis techniques make unsound optimizations to be precise; which brings into question the veracity of the security decisions.
resulting from such analysis [Liv15].

### 3.2.2.2 Dynamic Program Analysis

The application of dynamic analysis counteracts the limitations of static analysis in terms of false positives, although it may also incur high false negatives. Dynamic analysis is generally performed by hooking into the relevant protected operation, and then tracking program behavior after the protected operation is performed. Although OS enhancements for user data security may use a similar approach, this section focuses on dynamic analysis targeted towards detecting abnormal or unwanted behavior in programs. We describe prior work on enforcement of security guarantees in the next section on OS enhancements.

Dynamic taint tracking [NS05] for detecting privacy leaks is a prime example of dynamic application analysis on modern operating systems (e.g., TaintDroid [Enc10], AppInspector [Gil11]). The primary limitation of such fine-grained approaches is their susceptibility to implicit flows in application code. One solution may be to supplement the loss of accuracy with static analysis that detects implicit flows. However, this approach has been shown to incur high false positives [Kin08]. SpanDex [Cox14] limits the bits leaked due to implicit flows by enforcing a threshold on the number of operations that can be performed with private data. However, the approach is specific to certain kinds of information (e.g., passwords, pin codes), and may not be generalizable to abstract data.

Malware detection is another area that has leveraged dynamic analysis of applications [Kim08; Bos08; Liu09b; Por10; Bur11; Sha11; YY12; Ras13]. While some prior approaches use the effects of the application on the system (e.g., energy consumption) for behavioral analysis [Kim08; Liu09b], others incorporates the application’s actions (e.g., API calls) in the behavioral signature [Bos08]. Some approaches (e.g., AppsPlayground [Ras13], ANDRUBIS [Lin14b]) incorporate multiple malware detection techniques (e.g., taint tracking, monitoring sensitive API and system calls) to detect malicious behavior or privacy leaks. Particularly, tools that use energy consumption or similar features (e.g., CPU usage) often fail due to high false positives from benign applications (e.g., games), and may be costly in terms of their on-device resource consumption. To overcome the limitation of resource overhead, Paranoid Android [Por10] demonstrates that malware analysis can be moved off-site (i.e., to a virtual machine on a server) with minimal network bandwidth usage.
Further, prior research also improves the accuracy and precision of behavioral malware detection by examining traces from multiple users [Bur11]. However, a general limitation of dynamic malware detection is that malware may evade detection by not triggering its malicious behavior in an emulated environment [Jin14; VC14].

3.2.2.3 Operating System Enhancements

A diverse array of novel security enhancements have been proposed for Android. Lessons from these proposals have motivated changes to the Android OS, the most relevant of which are the run-time permission model introduced in Android 6.0, and the incorporation of SEAndroid [SC13] into the Android open-source project (AOSP). We now describe distinct categories of such enhancements, starting with modifications to the permission framework.

Enhancements to the Permission Framework: Prior work has built enhancements around Android’s permission framework, some of which motivate the need for run-time, fine-grained, control over permissions. For instance, CRePE [Con10] extends permissions to enforce user-specified context-specific security policies (e.g., restricting Bluetooth access in untrusted locations). Apex [Nau10] allows users to change application permissions at run-time, and even to dynamically limit the number of instances of permission use. However, it is well-known that fine-grained permission control and the resultant denials may break applications. To compensate for this trade-off, prior research has demonstrated the effectiveness of substituting fake information in protected API calls by applications, instead of denying access (e.g., MockDroid [Ber11], AppFence [Hor11], TISSA [Zho11]).

Transitive Access Control: Our work is not the first to explore transitive access control on Android. IPC Inspection [Fel11b] seeks to mitigate the latter deficiency by reducing permissions similar to Biba low watermark [Bib77]. However, this changes the semantics of permissions and can lead to permission bloat in applications. Quire [Die11] also seeks to mitigate confused deputy attacks [Har88] in Android. However, it takes an opposite stance and provides IPC provenance records that allow developers to manually evaluate request origins. Prior research also provides transitive protection for private data. Taint-Droid [Enc10] detects private data leaks via fine grained taint tracking on Android, but does not label processes, and hence is vulnerable to implicit flows or flows through native code. CleanOS [Tan12] and Pebbles [Spa14] use fine grained taint tracking on memory and
storage to evict and manage private data respectively. For tracking data in databases, the approaches require modification to the database library and are susceptible to false positives if a row contains data from many sources. Chapter 6 describes an alternate enforcement approach that is agnostic of the database library. Additionally, as we described previously on Section 3.1.4, existing DIFC proposals for Android [Jia13; XW15] offer interesting policy models, but face numerous trade-offs between security and practicality. The lessons learned from these prior models influence our work in Chapter 6.

**Containers:** The growing demand for smartphones in the enterprise has led to the requirement for adequate protection for user as well as enterprise data. The separation of work and personal environments into has been proposed as a solution, in order to protect work data from the user’s personal applications. For instance, approaches such as Samsung Knox [Sam13] and Android for Work [Anda] protect enterprise data secrecy by isolating groups of applications into different containers. However, containers cannot compensate for the lack of data secrecy guarantees, as they do not address threats within the container, i.e., the accidental export of secret data by a trusted application or the potential compromise of a trusted application. Virtual phones (e.g., Cells [And11]) are similarly inadequate for addressing the requirement of data secrecy, without isolating each application into its own separate container.

**Separation of Ad Libraries:** Preventing ad libraries from misusing application privileges and gaining access to private user data requires privilege separation [Pro03] between application code and ad libraries. To address this problem, Pearce et al propose AdDroid [Pea12], a modified Android framework that eliminates the requirement for the ad library to be included with the application. In AdDroid, Android’s API is extended to support advertising, and performs the functions of relaying ad metadata from applications to ad networks, and fetching as well as displaying ads to the user. AdSplit [She12] and AFrame [Zha13] take a different approach, by separating the executing context of library code to a separate process. However, while AdSplit layers the ad and content activities on top of each other, AFrame separates the UI into frames, with the ad library executing in a separate activity frame. Unlike prior approaches, LayerCake [RK13] allows applications to embed the ad library display into their activity, while also isolating it in a different process.

**MAC Frameworks:** Recently, Google adopted Security Enhanced (SE)Android [SC13] into
AOSP, to facilitate fine-grained MAC enforcement. SEAndroid enables SELinux for kernel-level MAC enforcement, and building a set of extensions for the middleware layer permission enforcement. Presently, SEAndroid provides fine-grained MAC enforcement for user space system daemons and system services (e.g., the Activity Manager). However, it only provides coarse-grained protection to third-party applications, as the policy administrators cannot specify the security requirements of the unknown interfaces and abstract data belonging to third party applications. Similarly, FlaskDroid extends Android security by making SELinux-style Type Enforcement (TE) available for third-party applications to protect their resources.

**Access Control Extensibility:** A majority of the access control frameworks described previously were implemented by extending a finite set of access control hooks present in the Android kernel or middleware. To facilitate the development, experimentation and deployment of such frameworks, we previously proposed the Android Security Modules (ASM) [Heu14] framework. ASM provides a set of authorization hooks that security enhancements (i.e., modules) can register for. When an authorization hook is triggered as the result of a protected event, ASM calls the security module(s) for the access control decision. ASM supports traditional hooks with binary (allow/deny) semantics, as well as hooks that allow fake information to be returned to applications. Similar to ASM, the Android Security Framework (ASF) [Bac14] also provides a programmable interface and authorization hook callbacks. A key difference is that ASM preserves existing security guarantees of the Android model by allowing the module to be restricted by permissions. On the contrary, ASF completely trusts the module writer, even allowing her to load a kernel module.

**Inlined Reference Monitors (IRMs):** The deployment of the OS security extensions described previously would require the cooperation of the OS vendor (e.g., Google). That is, while implementing the reference monitor in the OS provides strong security guarantees, such approaches they are rarely adopted in practice as they require extensive modifications to the platform. In response, researchers have proposed Inlined Reference Monitors (IRMs) [Erl04], i.e., inlining the security checks and enforcement in application code for fine-grained policy enforcement (e.g., Aurasium [Xu12], AppGuard [Bac13], RetroSkeleton [DC13], Capper [ZY14]). However, as IRMs share process address space with the application being sandboxed, they cannot inherently be said to be secure. This status quo has
been recently altered with the incorporation of the isolated process abstraction into the Android OS. An application can assign untrusted application components to execute in an isolated processes, which is devoid of any privileges, and under complete control of the application. This abstraction has recently been exploited to deliver inline reference monitors with stronger, OS-backed security guarantees in Boxify [Bac15]. While IRMs are still constrained by other limitations such as their incompatibility with application updates, the feasibility of extending existing abstractions such as the isolated process to enforce stronger properties (e.g., secrecy and integrity) needs further exploration.

3.3 Policy Specification

In this section, we describe prior work on automatic customization of privacy and security policies, which is closest to our contributions described in Chapter 7. Additionally, we discuss the complementary areas of languages for expressing security policies, enforcement of security policies, and the usability of policy interfaces.

3.3.1 Automatic customization of privacy policies

The proposals for user-controllable learning of security and privacy policies [Kel08; Cra11] are closely related to our vision of a security policy assistant. Simply stated, user-controllable learning allows the user to actively participate in policy customization. However, while our objective is to specify policies for abstract user-specific data objects (e.g., the user's scanned documents), prior work on user-controllable learning specifies policies for well-known private data or security preferences (e.g., Location data). Therefore, while our work cannot make any assumptions about the properties of data to be protected, prior work on user-controllable learning leverages well-known properties of data for predicting policies. For instance, Cranshaw et al. [Cra11] use a probabilistic model for prediction of location privacy policies, with the underlying assumption that a large number of data points would be available since location and time are continuous variables. Our approach of treating data or conditions as abstract objects is imperative for policy specification for user-specific data.

Further, machine learning has frequently been used to tag user content for access
control [Vya09; Squ11; Sin13; Bus15; Squ15]. For instance, Sinha et al. [Sin13] predict privacy preferences for Facebook users by classifying Facebook posts into various Facebook sharing policies. These prior works are complementary, as their content recognition techniques can be leveraged to automatically tag content for our proposed Policy by Example (PyBE) approach. Applications such as Google Photos use similar methods to tag content [Goo], and may also be used for PyBE.

Prior work in configuring mobile privacy settings states that even if a one-size-fits-all configuration is infeasible, a small set of privacy profiles may still be helpful [Liu14; Lin14a]. For example, Liu et al. study the configuration of 12 Android permissions across 4.8 million users, and cluster the permission settings into a small number of privacy profiles that apply to large populations [Liu14]. The approach works for Android permissions as every user configures the same Android permissions, although differently. However, user data is abstract and user-specific, and so is its security value to the user. Hence our user-specific approach may be more appropriate for user data.

Finally, Fang and LeFevre propose a “privacy wizard” for setting social network policies [FL10]. The approach uses active learning to specify data sharing policies with respect to the user’s friends. A significant difference is that the active learning approach used by the privacy wizard assumes independence among attributes of data points (i.e., friends). A similar assumption may not be appropriate for the data involved in user data scenarios. That is, the policy decision for a scenario formed by combining tags may be a property of all the tags combined. Therefore, our kNN-based approach is more appropriate, as it does not assume attribute independence.

### 3.3.2 Languages for expressing security policies

To protect user data held by applications, prior research has provided application developers with tools for expressing their security and access control policies [EK08; HE08; Har10; Sla14]. Efstathopoulos and Kohler [EK08] provide a policy description language for applications, to describe their policy for the Asbestos decentralized information flow control (DIFC) system [Van07]. The policy expressed by developers is compiled into low-level Asbestos primitives. Boat [HE08] provides similar support for generating policy for the Flume DIFC model [Kro07], while Harris et al. [Har10] automatically instrument programs with DIFC
policy. Further, Slankas et al. [Sla14] aid the developer by extracting access control rules from application-specific text artifacts using natural language processing (NLP). Such prior work provides developers with the means to compartmentalize the application and implement correct access control policies within the application. While these measures may indirectly benefit the user, they do not address the lack of policy specification from the user’s perspective.

3.3.3 Enforce policy

In this paper, we allow the specification of policies using example scenarios that may involve more than one tag, i.e., may indicate information derived from multiple sources, or the existence of multiple conditions. A large body of research in access and information flow control enforces the propagation of tags to enable labeling of derived content [BL76; Cum87; ML97; Van07; Kro07; Alj12; NE13; XW15; Bau15; Nad16]. For instance, Aquifer [NE13] and the approach by Jia et al [Jia13] provide policy models for enabling the data secrecy on Android. Maxoid [XW15] and Weir [Nad16] extend these prior works by making enforcement of such guarantees practical on Android. Further, Bauer et al [Bau15] provide a policy and enforcement model for information flow control on the Chromium Web browser. Prior work also motivates and enforces the policies similar to the network export control policy targets used as examples in this paper [Nad16; Ste14]. This paper advances the deployment of these novel systems by making their policy specification feasible.

3.3.4 Usability of policy interfaces

Johnson et al. [Joh10b; Joh10a] propose an interactive policy authoring template for usable policy specification, and provide a GUI-based prototype for generating policy templates. A prototype of our approach for a computing device may adopt the “interactive dropdowns” provided in the authoring template to ease the task of specifying initial examples.

Further, Reeder et al. [Ree08] propose “expandable grids” to visualize policy rules. A notable aspect of the work is that expandable grids enable the user to focus on the integrated effect of multiple policy rules, rather than having to reason about the interactions between
different rules through manual analysis. Such an approach may be adapted for visualizing policy examples as well, or even for displaying the predictions generated by our approach to the user.

Finally, access control gadgets (ACGs) [Roe12] provide a usable method for acquiring the user’s permission for user-owned resources on modern operating platforms (e.g., Android, iOS, Windows 8), i.e., by embedding the resource access permission into the user’s natural UI flow of accessing the resource. In a way, ACGs extend prior work that deals with involving the user in access control decisions for files [SE08; Sti06]. ACGs make traditional offline policy specification obsolete, but only for specific coarse-grained resources. That is, while ACGs may be feasible for a few coarse-grained security permissions, designing gadgets for an exponential number of personal data-use scenarios may not be feasible.
CHAPTER 4

A PROGRAMMABLE FRAMEWORK FOR EXTENDING ANDROID SECURITY

Consumer operating systems are changing. Modern commodity OSes such as Android, iOS, and Windows 8 place a high priority on the user-application experience. They provide new abstractions for developing user-applications: applications fill the screen; have complex lifecycles that respond to user and system events; and use semantically rich OS provided application programming interfaces (APIs) such as “get location,” “take picture,” and “search address book.” The availability of these semantically rich OS APIs vastly simplifies application development, and has led to an explosive growth in the number and diversity of available applications.

As the OS abstracts user and device resources to applications, it is also responsible for protecting users from misuse of the semantic API. These functional requirements have caused OS designers to rethink security, and have exposed new security challenges unique
to these systems. As described in previous chapters, modern OSes treat applications as security principals, assigning each application to a unique protection domain. However, the user may cause applications to share information, which exposes new data secrecy risks due to user-directed sharing. Similarly, another unique security challenge is the mediation of private user information such as contacts, location and phone identifiers. The OS mediates access to a large array of private data (e.g., contacts, location, phone identifier), while the decision of granting or denying access to applications rests with the user. Applications with access to such private data may misuse it to violate the user’s privacy. However, denying access may cause applications to malfunction. This conundrum is a unique challenge for systems that mediate access to high-level resources such as private user information.

Due to the numerous challenges of application-based commodity operating systems, the security research community has contributed significant discourse on the right security architecture for these new operating systems. Android has been the focus of this discourse, mostly due to its open source foundation, widespread popularity for mobile devices, and the emergence of malware targeting it. In the relatively short period of time since the Android platform’s initial release in 2008, there have been more than a dozen proposals for new Android security architectures [Enc09a; Ong09; Enc10; Nau10; Con10; Bug11b; Bug11a; Zho11; Ber11; Hor11; Fel11b; Die11; Bug13; NE13; SC13]. As we discuss in this chapter, while these security architecture proposals have very diverse motivations, their implementations often share hook placements and enforcement mechanisms.

The primary goal of this chapter is highlight the various security problems exposed by prior research on modern OS security, and to promote OS security extensibility [Wat13] in the Android platform. History has shown that simply providing type enforcement, information flow control, or capabilities does not meet the demands of all potential OS customers (e.g., consumers, enterprise, government). Therefore, an extensible OS security interface must be programmable [Wat13]. What makes this task interesting and meaningful to the research community is the process of determining the correct semantics of authorization hooks for this new OS architecture.

In this chapter, we describe the rationale behind the development of the Android Security Modules (ASM) framework, and the semantics of its authorization hooks that can be extended for building a diverse range of reference monitors. We derive these hooks by surveying over a dozen Android security enhancements. More importantly, we identify hook
semantics that not only support general access control requirements (i.e., allow/deny), but also address the unique security challenges of modern platforms (i.e., modifying data returned to apps, hooks in third-party applications).

### 4.1 Design Goals

A secure operating system requires a reference monitor [And72]. Ideally, a reference monitor provides three guarantees: complete mediation, tamperproofness, and verifiability. We seek to provide a foundation for building reference monitors in Android. A programmable interface for building new security enhancements to the Android platform is a medium to establish that foundation. The ASM design is guided by the following goals.

**G1** *Generic authorization expressibility.* We seek to provide the reference monitor interface hooks necessary to develop both prior and future security enhancements for Android. Not all authorization modules will use all hooks, and hooks may need to be placed at different levels to obtain sufficient enforcement semantics.

**G2** *Ensure existing security guarantees.* Android provides sandboxing guarantees to application providers. Allowing third-parties to extend Android’s security framework potentially breaks those guarantees. Therefore, ASM’s reference monitor interface hooks should only make enforcement more restrictive (e.g., fewer permissions or less file system access). Note that by only allowing more restrictive enforcement, we lose expressibility (e.g., for capability models).

**G3** *Protect kernel integrity.* As an explicit extension to Goal G2, we must maintain kernel integrity. Some authorization modules will require hooks within the Linux kernel. We cannot provide the LSM interface to third-parties without some controls. We explore several methods of exposing this functionality in Section 4.4.2.4.

**G4** *Multiple authorization modules.* While there have been proposals for supporting multiple LSMS [Sch13], official support for multiple authorization modules in Linux has not been adopted at the time of writing. We see benefit in allowing multiple ASM modules (e.g., personal and enterprise) and seek to design support for multiple
Authorization modules into the design of ASM. Achieving multiple authorization modules requires carefully designing the architecture to address potential conflicts.

### 4.2 Authorization Hook Semantics

The underlying motivation of ASM is to provide a programmable interface to extend Android security. Recently, Google adopted the UNIX-level portion of the SEAndroid [SC13] project into AOSP. However, Android security is significantly more complex than simply mediating UNIX system calls. Nearly all application communication occurs through Binder IPC, which from a UNIX perspective is an *ioctl* to `/dev/binder`. Mediating the higher level application communication has been the focus of most Android security research. The goal of this section is to explore these different proposals to identify a common set of authorization hooks semantics. That is, we seek to satisfy Goal G1 by surveying existing proposals to enhance Android security.

Academic and industry researchers have proposed many different security enhancements to the Android OS. These enhancements have a wide range of motivations. For exam-

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MockDroid [Ber11]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XManDroid [Bug11b]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TrustDroid [Bug11a]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FlaskDroid [Bug13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CRePE [Con10]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XManDroid [Bug11b]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TrustDroid [Bug11a]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FlaskDroid [Bug13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CRePE [Con10]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XManDroid [Bug11b]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TrustDroid [Bug11a]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
ple, Kirin [Enc09a] places constraints on permissions of applications being installed. Frameworks such as Saint [Ong09], XManDroid [Bug11b] and TrustDroid [Bug11a] focus on mediating communication between components in different applications. FlaskDroid [Bug13] and the aforementioned SEAndroid [SC13] project also mediate component interaction as a part of their enforcement. Our work Aquifer [NE13] enforces information flow control policies that follow the user’s UI workflow. IPC Inspection [Fel11b] and Quire [Die11] track Android intent messages through a chain of applications to prevent privilege escalation attacks. TaintDroid [Enc10] and AppFence [Hor11] dynamically track privacy sensitive information as it is used within an application. APEX [Nau10] and CRePE [Con10] provide fine-grained permissions. TISSA [Zho11], MockDroid [Ber11], and AppFence [Hor11] allow fine-grained policies as well as allow the substitution of fake information into Android APIs. While these proposals have diverse motivations, many share authorization hook semantics.

Table 4.1 classifies this prior work by authorization hook semantics. Nearly all of the proposals modify Android’s Activity Manager Service (AMS) to provide additional constraints on Inter-Component Communication (ICC). The Package Manager Service (PMS) is also frequently modified to customize application permissions. Permissions are also occasionally customized by modifying the interfaces to device sensors and system content providers containing privacy sensitive information (e.g., address book). Several proposals also require authorization hooks for file and network access, which are enforced in the Linux kernel.

The table also denotes two areas that are nonstandard for OS reference monitors. The first hook semantics is the use of fake data. That is, instead of simply allowing or denying a protected operation, the hook must modify the value that is returned. This third option is often essential to protecting user privacy while maintaining usability. For example, the geographic coordinates of the north pole, or maybe a coarse city coordinates can be substituted for the devices actual location. Replacing unique identifiers (e.g. IMEI or IMSI) to combat advertising tracking is a further example. The second interesting hook semantics is the inclusion of third-party hooks. That is, a third-party application wishes the OS reference monitor to help enforce its security goals.

Finally, TaintDroid [Enc10] and AppFence [Hor11] use fine-grained taint tracking. They modify Android’s Dalvik environment to track information within a process. However, dynamic taint tracking has false negatives, which may lead to access control circumvention.
It also incurs more performance overhead than may be tolerable for some environments. In this work, we only consider mediation at the process level. Therefore, TaintDroid and AppFence cannot be built on top of ASM. However, this does not preclude researchers from combining TaintDroid with ASM.

### 4.3 Architectural Overview

The authorization hooks identified in the previous section describe semantically what to mediate, but not how to mediate it. Existing Android security enhancements define hooks in different ways, not all of which provide correct or complete mediation. ASM provides a reference monitor interface for building new reference monitors. Reference monitor developers only need to focus on their core security policy logic, while ASM performs the tasks of hook placement and enforcement. ASM separates basic enforcement tasks from the policy logic, paving the way for the separate scrutiny of the common set of authorization hooks and their placement in the OS.

Figure 4.1 shows the architecture of the ASM framework. The ASM Bridge receives access control callbacks (i.e., protection events) from authorization hooks placed at all levels of the Android OS. ASM allows reference monitors to be deployed as separate modules, called *ASM apps*. Each ASM app registers a set of authorization hooks it requires, and implements a callback for each hook. When a protection event occurs, the ASM bridge automatically invokes the callback in each ASM app that has registered for the hook. Note that for performance reasons, a hook is in a “disabled” state until at least one ASM app registers for it, and is not invoked. For hooks in the Linux kernel, the ASM LSM performs...
4.4 Authorization Hooks

We identify our initial set of authorization hooks based on the security problems addressed by the reference monitors discussed in the previous section. ASM’s reference monitor interface provides authorization hooks placed throughout the Android OS. We begin by describing the hooks that return fake data, followed by the five general categories of hooks, namely (1) lifecycle hooks, (2) OS service hooks, (3) OS content provider hooks, (4) third-party app hooks, and (5) LSM hooks.

4.4.1 Callbacks Modifying Data

While general access control requirements are limited to allow/deny decisions, some scenarios may require the reference monitor to return fake information. For example, MockDroid [Ber11] modifies values (e.g., IMEI, location) returned by OS APIs before they are sent to applications. ASM supports data modifications by providing a special hook type.

The reference monitor interface exposes two variants of each authorization hook that may require data replacement; a normal variant that allows the corresponding callback to only allow or deny an event, and a modify variant that allows the callback to modify the value returned by the OS API or content provider, in addition to allowing or denying the access. Note that if a consensus-based policy is used to reconcile the decisions of multiple security modules, a denial by any one module will cause the access to be denied. In this specific scenario, invoking normal callbacks before the modify callbacks carries a significant
Figure 4.3 A lifecycle hook that allows modified data to be returned to the receiving application.

Example 1: Figure 4.3 shows examples of the two hook variants for the start_activity protection event, which may require data modification. The first prototype, start_act(), is the ASM Bridge callback in Android’s Activity Manager Service. This protection event occurs after the activity component to be started is known (i.e., resolved), but before it is actually started. The ASM Bridge splits the hook into its normal and modify versions. As a means of protecting the integrity of the fields solely used by the system, ASM apps may only modify the fields used by the third party app receiving the intent (i.e., the extras) via the modify callback (i.e., start_act_mod). To ensure this requirement, the modify callback includes a mutable Bundle containing the extras for the ASM app to modify, while the original intent is made immutable. The ASM bridge replaces the extras in the intent with the edited extras before returning the callback to the Activity Manager service.

4.4.2 Lifecycle Hooks

As previously described in Chapter 2, Android’s Activity Manager service and its subsystems control application and component lifecycles. ASM controls numerous lifecycle events through hooks in the Activity Manager, as well as the Package Manager service for a minority of the hooks. Hooks in this category include, but are not limited to: resolving intents, starting activities and services, binding to services, registering for broadcast receivers, and receiving intent broadcasts. The start_activity hook described in Figure 4.3 is an example of a lifecycle hook.
4.4.2.1 OS Service Hooks

Android exposes numerous OS APIs for access to high-level resources, such as the user's GPS location, contacts, IMEI and the camera. These APIs are implemented in various system services in the OS middleware (e.g., location and telephony services). ASM allows ASM apps to mediate these protected APIs via hooks in system services.

Example 2: Figure 4.4 shows the callback for the getDeviceId() OS API call, which returns the device identifier (i.e., the IMEI). This API is implemented in the telephony service (i.e., the PhoneSubInfo class). This hook has a modify variant as well, which may return a fake IMEI value to the application. ASM apps receiving this callback may return an allow or deny, or may allow with a modified (i.e., fake) value placed in the first index of the device_ids array in the getDeviceId_mod() callback. For some hooks, ASM leverages Android’s AppOps subsystem. The scope of this chapter is a description of the diverse array of security enhancements and the common security hooks that may fulfill them; for a detailed explanation of the AppOps integration, we refer the reader to the full paper of the ASM framework [Heu14].

4.4.2.2 Content Provider Hooks

As described previously in Chapter 2, content provider components are daemons that provide an interface to access and share data with other applications. On Android, the OS defines some content providers for storing well-known user information, such as contacts, calendar information, and call logs. We define hooks in these OS-defined content providers (e.g., Calendar, Contacts, and Telephony), and create separate authorization hooks for insert, update, delete, and query functions. Note that the hooks for insert, update and delete functions are invoked before the action is performed, in order to preserve the integrity of the provider’s data. On the contrary, the hook for the query is invoked after the query has
executed at the provider's end, but before the query result is returned to the application, in order to filter the query result.

4.4.2.3 Third Party Hooks

Applications often bring secret user information to the device (e.g., secret email attachments). However, existing security mechanisms (e.g., Android permissions) are insufficient for providing fine-grained enforcement for application resources. Additionally, many of the security challenges that motivate novel security extensions for Android (e.g., the need to return fake private data) may apply to data objects and interfaces of third-party applications. As a result, ASM allows third-party Android applications to dynamically add hooks to its reference monitor interface. These third-party hooks provide a valuable medium for extending enforcement into third-party applications that may require protection for sensitive resources. The complete identity of a third-party hook is composed of its name and the package name of the application that adds the authorization hook to the ASM interface, as well as its own code (i.e., before the respective protection event). For security reasons, third parties do not specify their package name; ASM obtains it using the registering application’s uid received from Binder.

As shown in Figure 4.5, ASM apps implement two generic methods for receiving callbacks to receive callbacks for third-party hooks. As in the case of other hooks, one method handles a normal allow/deny callback, while another addresses the need for data modification. A generic Bundle object is used to capture the access control decision; i.e., the generic object is passed from the authorization hook to the ASM Bridge via the callback, and then to the registered ASM apps.

As third-party hooks are created dynamically, ASM apps receive hook callbacks for all
their registered third-party hooks via a single, generic, interface. The ASM app must identify
the hook by name, and interpret the data accordingly (i.e., using the arguments expected
from invocation of the hook). The ASM app processes the hook, and returns a decision
along with a modified data value, if required.

4.4.2.4 LSM Hooks

For complete mediation of accesses to standard OS objects such as files, network sockets
and pipes, authorization hooks must be implemented in the OS kernel. For this purpose,
ASM extends the LSM reference monitor interface in the kernel.

We define a special ASM LSM that implements LSM hooks and performs synchronous
upcalls to the ASM Bridge to complete the access control decision. Kernel upcalls are con-
sistent with the activation logic controlling the rest of the hooks, i.e., an upcall is only made
when at least one ASM app has registered for the hook. Our implementation integrates ASM
LSM without disabling SEAndroid (Goal G2), through the use of a multi-LSM patch [Sch13]
for the Linux kernel. While we implement ASM authorization hooks for the most commonly
used LSM hooks (e.g., file_permission, socket_connect), more hooks can be added
with nominal effort.

4.5 Summary

In this chapter, we study over a dozen security enhancements for Android, understand
the unique security challenges they address, and discover their novel authorization hook
requirements. In doing so, we not only discover the new security challenges exposed for user
data on modern commodity operating systems, but also discover that most such challenges
can be addressed using a common set of authorization hooks. We propose the Android
Security Modules framework as a programmable interface for extending Android security.
Although similar reference monitors have been proposed for Linux and TrustedBSD, ASM is
novel in how it addresses the semantically rich OS APIs provided by new operating systems
such as Android. Of particular note is the ability for hooks to replace data, as well as for third-
party application developers to define new hooks. We conclude the chapter by describing
the semantics of the reference monitor hooks developed as a part of designing ASM.
Operating system security architectures are currently undergoing a fundamental change. Modern OSes [Roe12; Wan07], such as Android, iOS, and Windows 8, take the suggestion of decades of security research [Wic90; Ioa02; ST05; Enc08] and run each application as a unique security principal. While having finer-grained security principals prevents many obvious attacks, complete sandboxing [Gol96] is inadequate.

Applications share data with one another, perhaps more so now than in the past. Consider the Android platform where applications are designed to work together to perform a larger, user-defined task. For example, a shopping app might: 1) invoke a barcode scanner app that uses the camera to read the UPC from an item, 2) look up that item on the Web,
and then 3) use a social networking app to share the item and best deal with friends. This modularity strikes a balance between simple UNIX tools (e.g., sed, grep) and monolithic GUI applications (e.g., MS Office).

A key challenge for modern OS security is controlling this user-directed workflow between apps and preventing accidental information disclosure. For example, a photo of a whiteboard containing meeting notes might be inadvertently uploaded to a social networking site, or a confidential document might be inadvertently stored on a cloud server when viewed. Accidental disclosure is growing concern for consumer privacy, and has been a large concern for companies and organizations attempting to comply with the many data security compliance standards, e.g., HIPAA [US 02], GLBA [US 99], PCI DSS [Pay10], and IRS 1075 [US 10].

Preventing accidental disclosure is not as simple as restricting the set of applications an application with sensitive data can interact with (e.g., Saint [Ong09]). A trusted application receiving data might share that data with another application that has unexpected disclosure. Hence, in the collaborative application environment, we must address the accidental disclosure problem as one of information flow. Specifically, we identify the data intermediary problem as a growing concern for modern OSes. The data intermediary problem is a subtype of the secure information flow vulnerability that results when user choices dictate data flows between user-facing apps and apps lose control of the data.

In this chapter, we present Aquifer as a policy framework and system to mitigate accidental information disclosure in modern operating systems. Aquifer is specifically designed to protect large, application-specific, user data objects such as office documents, voice or written notes, and images. In Aquifer, developers of applications that originate data objects specify secrecy restrictions based on the runtime context and the purpose of the app. This policy restricts all apps participating in a user interface workflow that Aquifer dynamically constructs as the user navigates different applications. Aquifer enforces two types of secrecy restrictions: export restrictions ensure only specific apps can export the data off the host, and required restrictions ensure that specific apps are involved in workflows when exporting controlled data objects read from persistent storage. This policy is specified using a decentralized information flow control (DIFC) motivated language that allows many data owners on a workflow to participate in secrecy restrictions. In effect, Aquifer allows applications to gain control of shared sensitive data, thereby addressing the
Data intermediary problem for these large data objects.

This chapter makes the following contributions:

- We identify the data intermediary problem as a growing concern for modern operating systems. While the data intermediary problem is present in traditional commodity OSes, the lack of application separation did not expose it as a concern.

- We propose the Aquifer policy framework for addressing accidental disclosures that result from the data intermediary problem in modern OSes. Aquifer allows app developers to contribute DIFC-based secrecy restrictions to protect application-specific data objects. We formally define the policy logic and prove its safety.

- We provide a proof-of-concept implementation of Aquifer and integrate it with Android. We demonstrate how Aquifer can be practically realized within an existing platform, and provide three case studies by modifying popular open source applications.

5.1 The Problem of Accidental Data Disclosure

Modern operating systems such as Android and Windows 8 present a new programming abstraction for software developers. Instead of placing all functionality into a single window with multiple dialog boxes, the application's user interface is separated into multiple screens where each screen handles a specific task. To complete a task, the user navigates through a series of screens. These screens may be in the same or different applications. For example, Android applications use intents addressed to action strings (see Section 2.2) to help the OS find the best application for a task. Similarly, Windows 8 provides “share charms” to help users complete tasks with different applications. Finally, iOS provides limited sharing and navigation between applications using URL protocol handlers.

In each of these OSes, applications are treated as separate security principles, although the specific security mechanisms differ. Android separates applications as different UNIX user IDs, and Windows 8 uses SUIDs. In contrast, iOS runs all applications as the mobile user with a generic sandbox policy. However, digital signatures are used to identify applications, and permission state (e.g., location access) is saved per-application.
Throughout the remainder of the chapter, we frequently use Android to simplify discussion and provide concrete examples. Our choice of Android is motivated by several factors. Most importantly, Android provides the most flexible sharing model between applications. As the following discussion will make clear, sharing data between applications underlies the security problem. Android is also open source, used by hundreds of millions of consumers, and well described in security literature. We believe that other modern OSes that provide clear sharing abstractions (e.g., share charms in Windows 8) can benefit from our policy abstractions and design; however the implementation details will differ.

5.1.1 Use Case: Signing a Document

The following example provides a simple use case of how a user Alice might physically sign a document using several applications in a modern OS. Note that this is just one of many potential ways Alice can execute this task.

Alice receives a confidential contract in her business Email app. She needs to sign and return the contract, but does not have access to a printer or a scanner. Therefore, Alice uses the DocuSign app on her smartphone to digitally attach a written copy of her signature. The task begins by Alice accessing the message containing contract.doc in the Email app. Alice reads contract.doc by sharing it with the DocuView app. After reading contract.doc, Alice wishes to sign it with DocuSign; however, DocuSign only operates on PDF files. Therefore, Alice shares contract.doc with the WordToPDF app to create contract.pdf, which returns the PDF to DocuView. Alice then shares contract.pdf with DocuSign, which embeds a copy of her written signature, creating signed.pdf. The file is then shared with the Email app to return the signed contract via Email. This task workflow is depicted in Figure 5.1.

5.1.2 Problem Definition

The document signing use case provides an example of how a user might combine several applications to accomplish a task. In the example, the business Email app received a confidential contract. Based on the email headers, Email knows contract.doc should not be exported off of the host by any application except itself. However, Alice needs to
modify contract.doc in ways that Email does not support. One of the valuable features of modern OSes is the large collection of third-party applications that act as modules to perform specific tasks. While these apps provide valuable functionality, they also present a security risk: once Email shares contract.doc with another app, it loses control of it, which may result in accidental disclosures that violate compliance regulations (e.g., HIPAA [US 02], GLBA [US 99], PCI DSS [Pay10], and IRS 1075 [US 10]). For example, the WordToPDF application might perform the PDF conversion on a cloud server, or DocuView might synchronize viewed documents with cloud storage. Similarly, signed.pdf containing the user’s written signature should only be used when the user intends. The user may be unaware (or not think of) the sometimes subtle implications of selecting which apps to use.

The preceding example demonstrates the data intermediary problem. This problem occurs whenever the user directs an application to share sensitive data with another application that may not be trusted with that data. From the Email app’s perspective, all of the other applications are data intermediaries in performing the user’s task of signing contract.doc. We have created a specific term for this subproblem to differentiate it from secure information flow problems that result from background processing. The data intermediary problem is specific to information flows that result from user choices in selecting applications to process data. Furthermore, the problem is most apparent in modern OSes, because they 1) distinguish applications as security principals, and 2) provide modular applications to perform larger user tasks. We note that the data intermediary problem has always been present in operating systems; however, it made little sense to discuss when all user applications ran with the user’s ambient authority.
For the purposes of this chapter, we focus on the data intermediary problem with respect to accidental data disclosure that results from user selection. We leave the much harder threat model of a malicious application as the motivation for future work. However, we note that the primitives described in this chapter can form the basis of a system to defend against this stronger adversary, as we show in Chapter 6.

5.1.3 Threat Model

While our work is motivated by data security compliance regulations, we do not focus on the specific compliance rules themselves. Instead, we seek to address the broader challenge of creating mechanisms that help prevent the accidental disclosure portion of the data intermediary problem. We are specifically concerned with preventing the accidental export of large, application-specific, user data. There are potentially many data owners with different secrecy requirements. Therefore, an application may be both a data owner and a data intermediary, depending on the policy perspective, and each data owner's secrecy requirements must be met, even if doing so prevents data from being used.

Accidental data disclosure may occur in various ways. The user may share data with the wrong application (e.g., sharing a photo of whiteboard meeting notes via a social networking app). Such data export may not comply with the owner's policy, but may still occur through the user's interaction. Poorly programmed applications may also unknowingly leak private data to the cloud. For example, a document editor might backup documents to the cloud, and an app might send data as part of targeted advertisements.

The work in this chapter does not seek to prevent malicious data disclosure. That is, we do not address side channels or collusion between applications. We also do not consider malicious daemons that operate outside our confinement. Finally, we are specifically concerned with data on the host and do not address exposure of data from cloud services once it is allowed to leave the host.

5.2 Aquifer Overview

Aquifer is designed around the concept of a user interface workflow. As previously discussed, an emergent property of modern OS applications is that they are relatively simple, purpose
or service specific, and often combined with other apps to perform a larger task. When the user performs a task, the execution transitions between UI screens. The next UI screen can be in the same or different application. Aquifer tracks the specific instances of the UI screens used to perform the user’s task and abstracts them as a UI workflow.

Security policy is applied to the UI workflow abstraction, as shown in Figure 5.2. We choose the UI workflow abstraction to define security policy, because it approximates the task at hand. All operations performed as part of this task will have similar security requirements. Frequently, the task will be centered around a single data object and its derivative objects, as demonstrated in the document signing use case.

Note that UI workflows are not necessarily linear. They are dynamically defined as the user navigates functionality on the host. This includes branches to perform subtasks. For example, a user interacting with a shopping application may navigate to a barcode scanner to retrieve the UPC code of a product via a camera. When this branch returns, the user continues the task. As shown in Figure 5.2, Aquifer allows applications on the branch to contribute to secrecy restrictions (e.g., UI screen $D$).
Figure 5.3 depicts the Aquifer architecture for Android. Aquifer provides an API for applications to manage policy. This policy is enforced by the Aquifer System, which places hooks into Android's Activity Manager service. Finally, Aquifer has a small kernel component to monitor file communication.

Aquifer is built around the following principles:

**Decentralized policy specification:** Modern OSes increasingly contain application-specific data. Therefore OS providers cannot practically define security policy. Instead, Aquifer uses the multiple-owner policy semantics of decentralized information flow control (DIFC) [ML97]. Since each application is a potential stakeholder on data, DIFC provides a well-founded notion of data ownership and an articulation in each context of what each principal is trusted to do with that data.

**Developers & Users define policy:** The developers of applications that own data can frequently identify security sensitive data. Aquifer then infers user intention from the UI workflow. While this reduces the burden on the user, it does not entirely eliminate it. Sometimes the application must distinguish between confidential and public data. This context can frequently be acquired via preliminary labeling, which ranges by application. For example, the Email app in our use case could determine secrecy requirements from an Email header set by the sender. Applications such as note apps (e.g., Evernote) already have semantic tags on data (e.g., business, personal) that can be leveraged. User data labeling has been shown to be useful for specifying policy [Kle12]. In other cases (e.g., DocuSign), the policy specification is inherent to the functionality of the app.

**Compatibility with legacy applications:** Aquifer focuses protection on large, application-specific, data objects. Applications frequently process these data objects locally. This allows Aquifer to be compatible with most legacy applications and only requires modifications of applications that must specify policy (i.e., data owners). If no secrecy restrictions are specified, Aquifer uses a default-allow policy.

**Minimizing policy violations:** Policy violations confuse users by either prompting the user to make security decisions, or breaking functionality. Aquifer helps minimize policy violations by allowing applications to influence the functionality available to users. For example, Android uses “action strings” (e.g., ACTION_SEND, ACTION_VIEW, ACTION_EDIT) that help the OS find an appropriate consumer for shared data. When Android finds multiple
possible recipients, the user is presented a list of targets from which to choose. Similar functionality is provided by Windows 8’s share charm. If the user chooses a target application that attempts to export data, and the UI workflow export restriction denies the app to use the network, a security exception will result. Often, this will break the functionality of the app, resulting in a poor user experience. Therefore, to prevent such scenarios from even occurring, Aquifer allows a data owner to specify a UI workflow filter that limits the potential targets.

**Compatibility with background functionality:** UI screens may communicate with daemons (e.g., service and content provider components in Android). If interaction with a daemon passes sensitive data between two UI workflows, e.g., between screens C and Y in Figure 5.2, Aquifer must propagate the policy restrictions to the receiving workflow. However, Aquifer cannot simply propagate the workflow security policy to the daemon process, as this would cause the daemon and subsequent UI workflows to be restricted by all previous UI workflow policies. Ultimately this would result in an unusable system. Therefore, Aquifer requires a more precise method of tracking information within daemons. For this, Aquifer could leverage systems such as TaintDroid [Enc10] and CleanOS [Tan12]. However, the primary focus of this chapter is the ability to specify and enforce security policies with respect to the UI workflow. Therefore, for our prototype implementation, we use a lighter weight heuristic based on tracking file descriptors used by daemons (see Section 5.4.3).

### 5.3 Aquifer Policy

A key challenge of Aquifer is defining the appropriate policy semantics for addressing the data intermediary problem in modern OSes. We first motivate the security policy types supported by Aquifer and then formally define the logic.

#### 5.3.1 Policy Types

The primary concern of Aquifer is accidental export of high-value, application-specific user data. Therefore, our secrecy restrictions are defined with respect to export control. Export restrictions allow any functionality on the host, but prevent leakage to remote parties that
are not mediated by the framework. As mentioned in Section 5.2, Aquifer uses a default-allow policy to ensure compatibility with legacy applications processing unconstrained data objects. However, the policy becomes default-deny if restrictions are present.

Based on a manual survey of Android applications, we identified the need for the following secrecy restrictions.

**Export Restrictions:** The most basic type of secrecy restriction is a whitelist of applications that are allowed to send data off the device. Frequently, the whitelist contains only the application that specifies the export restriction. For example, in the document signing use case in Section 5.1, the *Email* app wishes to ensure that only it can send *contract.doc* and derivative files off the host. We allow an application to specify a list to support suites of applications or lists of known trusted applications.

**Required Restrictions:** The second type of secrecy restriction is motivated by copies of files left on persistent storage. Required restrictions ensure that cached copies of files cannot be later exported without the knowledge of the data owner. In our document signing use case, *DocuSign* may wish to protect the handwritten signature of Alice by ensuring that a file containing the signature can only be sent off the device when *DocuSign* participates in the workflow. Since *DocuSign* is the trusted authority for handwritten signature data, it trusts itself to ensure user approval for using a workflow that involves sending a signed document off the host. Required restrictions are particularly useful for applications that provide a UI for the user to choose and return a specific file. Finally, while it is likely that applications will only specify a single required restriction, Aquifer allows a list. We currently require all applications on the list to be present on the workflow. In the future, we will explore the usefulness of “*k of*” policies.

**Filters:** A direct consequence of enforcement of export restrictions is access control violations, and Aquifer attempts to reduce these violations through workflow filters. Aquifer allows applications to define these UI workflow filters specifically to enhance usability. In the case of Android, filters limit the results of intent resolution shown to the user. Similar filters can be constructed for Windows 8’s share charm.
5.3.2 Policy Logic

Aquifer formalizes the export, required, and filter policy types into a logic. Our logic is motivated by the decentralized label model (DLM) [ML97]. We chose DLM over other DIFC logics [Van07; Zel06; Zel08; Kro07; KT09; Ste11a] due to its clear owner semantics in the policy label. We extend DLM by replacing the set of readers with a tuple containing our export, required, and filter restrictions. Note that Aquifer uses DIFC to control data export and not interaction between applications.

Aquifer uses applications as security principals. We chose applications over UI screens, because the fine granularity of UI screens would be cumbersome to specify and manage. Developers defining security policy do not necessarily know the UI screens in other applications.

The UI workflow policy itself is a collection of owner policies, where each owner is an application. The owner policy contains an export list, a required list, and a workflow filter:

Definition 1 (Export list). An export list $E$ is a set of applications that may access the network while participating in the UI workflow.

Definition 2 (Required list). A required list $R$ is a set of applications that all must have been present on the UI workflow at sometime in the past for any application on the UI workflow to access the network.

Definition 3 (Workflow filter). A workflow filter $F$ is a set of tuples $\{(s_1, T_1), \ldots, (s_n, T_n)\}$, each containing an action string $s_i$ and a set of targets $T_i$. If the normal resolution of an intent message sent to action string $s_i$ is a set of apps $N$, then the resulting allowed target applications is $N \cap T_i$.

To simplify discussion, we define functions for retrieving the action string and set of targets from a workflow filter. For a filter $F$, $\text{actions}(F)$ returns the set of all action strings in $F$. Similarly, for a filter $F$ and an action string $s$, $\text{targets}(F, s)$ returns the set of target applications for action string $s$. Note that for the following logic to be correct, we assume that there does not exist an $s$ such that $\text{targets}(F, s) = \emptyset$. If this occurs, Aquifer simply removes $s$ from $\text{actions}(F)$, implying there are no restrictions for $s$ (default allow).

Having defined export lists, workflow filters, and required lists, we can now define a workflow label.
Definition 4 (Workflow label). A workflow label \( L \) is an expression \( L = \{ O_1 : (E_1, R_1, F_1); \ldots; O_n : (E_n, R_n, F_n) \} \), where \( O_i \) is an owner (application) and \( E_i, R_i, \) and \( F_i \) are an export list, required list, and workflow filter, respectively, specified by \( O_i \).

A label \( L \) contains a set of owners denoted \( \text{owners}(L) \), which is the set of all owners that have specified a restriction for the UI workflow (i.e., \( O_1, \ldots, O_n \) in Definition 4). To modify \( L \) (i.e., add, remove, or change), an owner \( O_i \) must contain the active UI screen and can only modify its portion of \( L \) (i.e., \( O_i \) cannot change \( E_2, R_2, \) or \( F_2 \)).

We define functions for retrieving the parts of an owner’s policy from a label \( L \). Care is needed to account for Aquifer’s default allow policy when no restrictions are specified by an owner. Let the set of all applications be \( A \), and the set of all possible action strings be \( S \). For each owner \( O_i \), \( \text{exports}(L, O_i) \) returns \( E_i \), unless \( O_i \notin \text{owners}(L) \) or \( E_i = \emptyset \), in which case \( \text{exports}(L, O_i) \) returns \( \mathcal{S} \). Semantically, this means \( O_i \) does not have any export restrictions. Similarly, for each owner \( O_i \), \( \text{filters}(L, O_i) \) returns \( F_i \), unless \( O_i \notin \text{owners}(L) \) or \( F_i = \emptyset \), in which case it returns \( \{ (s, \mathcal{S}) | \forall s \in \mathcal{S} \} \). In contrast, for each owner \( O_i \), \( \text{requires}(L, O_i) \) returns \( R_i \) regardless if \( O_i \) exists or if \( R_i \) is specified.

A useful concept is the effective policy. That is, given a label \( L \) with multiple owners, what policy should be enforced. We define the effective export list, required list, and workflow filter as follows.

Definition 5 (Effective export list). For a workflow label \( L \), the effective export list \( E_e = \bigcap \text{exports}(L, O), \forall O \in \text{owners}(L) \).

Definition 6 (Effective required list). For a workflow label \( L \), the effective required list \( R_e = \bigcup \text{requires}(L, O), \forall O \in \text{owners}(L) \).

Definition 7 (Effective workflow filter). For a workflow label \( L \), the effective workflow filter \( F_e \) is the set of tuples containing action string and corresponding target application set created by taking the union of all action strings and the intersection of the targets for those action strings. More precisely, \( F_e = \{ (s_i, T_i) | s_i \in \bigcup \text{actions}(F) \text{ and } T_i = \bigcap \text{targets}(F, s_i), \forall F \in \text{filters}(L, O), \forall O \in \text{owners}(L) \} \).

There are various scenarios in which Aquifer must combine two workflow labels, e.g., propagating a workflow label from a file, or through a daemon. When this occurs, we join
the two labels \( L_1 \) and \( L_2 \) to create a new label that is the least restrictive label that maintains all of the restrictions specified by \( L_1 \) and \( L_2 \) [ML97].

**Definition 8 (Label join \( \sqcup \)).** For workflow labels \( L_1 \) and \( L_2 \), the join \( L = L_1 \sqcup L_2 \) is a new label ensuring the following for all owners \( O \):

\[
\begin{align*}
owners(L) &= owners(L_1) \cup owners(L_2) \\
exports(L, O) &= exports(L_1, O) \cap exports(L_2, O) \\
requires(L, O) &= requires(L_1, O) \cup requires(L_2, O) \\
filters(L, O) &= \{(s_i, T_i) | s_i \in actions(F_1) \cup actions(F_2), \\
&\quad T_i = targets(s_i, F_1) \cap targets(s_i, F_2), \\
&\quad \text{where } F_1 = filters(L_1, O), \\
&\quad F_2 = filters(L_2, O)\}
\end{align*}
\]

Similar to the definition of an effective workflow filter, the last rule ensures that the workflow filter for the new label \( L \) contains the union of action strings in \( L_1 \) and \( L_2 \), and the intersection of the target applications for each of those action strings. Finally, we note that when the above conditions results in the universal set for one of the restriction lists, our implementation removes the list to indicate default allow.

### 5.4 Aquifer System Design

The Aquifer system enforces the Aquifer policy logic within a modern operating system. While we try to keep our description general, we frequently provide concrete examples using the Android platform.

#### 5.4.1 Managing UI Workflows

As described in Section 5.2, Aquifer defines and enforces policy with respect to a UI workflow. A UI workflow is a graph that tracks the history of UI screens that comprise the user’s task. This section discusses how Aquifer identifies and manages the workflow.
Identifying the Workflow: As the user navigates to a new UI screen (e.g., Android activity component instance), Aquifer adds the screen to the workflow. Aquifer does not need to store the exact workflow graph to enforce the workflow label policy. Aquifer needs to keep track of: 1) \( W_V \), a list of applications the workflow has visited (for effective required list \( R_v \)), and 2) \( W_R \), a list of metadata for currently “running” UI screens (for effective export list \( E_v \)). The metadata required for \( W_R \) is dependent on the specific Aquifer implementation and the information required to enforce the policy. For this discussion, we assume it contains at least the app name and process identifier.

Ideally, we would like to start each UI screen in a separate process. This allows Aquifer to easily enforce the workflow policy by turning network access on and off for the process. If the same process is used in two simultaneous UI workflows with labels \( L_1 \) and \( L_2 \), Aquifer must assign both workflows the label \( L_1 \cap L_2 \) in order to preserve the restrictions on both workflows. This can lead to overly restrictive policy.

At first, separate processes for UI screens seemed straightforward for our Android implementation of Aquifer. Android is designed to allow components to transparently interoperate with components in different processes. Therefore, conceptually we could modify the Android framework to start each activity component instance in a separate process. However, we ran into two problems. First, activity components are simply Java objects that extend the \texttt{Activity} class and sometimes share global variables with the rest of the application. In such cases, starting the activity component in a new process causes the application to crash when an uninitialized value of a global object is retrieved. Second, in the cases when activity components could be run in a separate process, Android did not provide an easy mechanism to start multiple processes if multiple instances of that activity component are needed.

To account for these limitations, we made the following compromise. When starting an activity component, Aquifer checks if the process for that component already exists. If not, a new process is started, and there is no problem. If a process does exist, Aquifer determines if it is part of the current UI workflow. If so, the activity is started in this process. If not, Aquifer terminates the process. If applications are developed following Android’s recommended conventions, an activity should save its state to persistent storage when Android calls the \texttt{onStop()} callback, indicating the activity is no longer visible. Aquifer then starts a new process for the activity for this UI workflow.
This approach is less desirable than poly-instantiation (suggested above), because if applications do not save their state, data loss may result. An undesirable user experience may also result if an activity component in the middle of a UI task is terminated, or if activities call each other in a loop. One way developers can reduce the impact of Aquifer's need to terminate processes is to develop their applications such that each activity starts in a separate process. This can be easily done using annotations in the app’s manifest file.

**Policy Administration:** Only the active (i.e., currently displayed) UI screen can modify the UI workflow policy. Aquifer exports a policy management API to applications that includes the ability to query, set, and remove the export list, required list, and workflow filter for that application. We note that a UI screen can only retrieve and modify the policy for the application that contains it. This keeps an application from reading the policies set by other applications, but it does not prevent it from learning the effective policy, which can be queried by testing network access.

**Removing Unrelated Policy:** In developing Aquifer for Android, we identified an opportunity to remove unnecessary restrictions from the UI workflow label $L$. Activity components can be started in two ways: `startActivity()` and `startActivityForResult()`. The former method never returns a value, whereas the latter does. Aquifer leverages this artifact by pruning the workflow label as follows. When UI workflow branch returns, Aquifer determines if the activity component was started for a result. If not, Aquifer checks if owner policy can be removed. An owner policy for application $O$ can be removed from $L$ if and only if: 1) no UI screen of app $O$ exists in the set of running UI screens $W_R$, and 2) no past UI screen (e.g., activity component instance) of app $O$ returned a value. To ensure the latter condition, we modified $W_R$ to include an extra bit of information indicating whether or not a UI screen for the application was started for a result. Note that this heuristic is conservative and may not remove an owner policy if a value was returned on a branch that later does not return a value. Once $W_R$ is empty, Aquifer terminates the workflow.

### 5.4.2 Enforcing Policy

The Aquifer UI workflow policy restricts which applications can send data to the network. The workflow label contains a list of owners and corresponding export lists, required lists, and workflow filters that are used to calculate the effective export list $E_e$, effective required...
Aquifer enforces $E_e$ and $R_e$ by controlling the network access of the process containing the UI screen. Since applications are security principals, it does not matter if each UI screen runs in its own process, or all UI screens for an application run in the same process. For each process $p$ corresponding to application $app(p)$, Aquifer enables network access if and only if:

$$(E_e = \emptyset \lor app(p) \in E_e) \land (\forall r \in R_e, r \in W_f)$$

Simply put, this equation implements default allow only if $E_e$ is empty and all $r$ in $R_e$ are satisfied. Otherwise, the application corresponding to $p$ must be listed in $E_e$.

Aquifer must re-evaluate the network access control for each process on a UI workflow whenever: a) an application on the UI workflow modifies its policy, or b) a new UI screen is added to the workflow. The latter condition is only necessary when the application for the added UI screen completes the restriction requirement for satisfying $R_e$.

Finally, as described in Definition 3, Aquifer enforces workflow filters by reducing the list of applications shown to the user on transitions between UI screens.

### 5.4.3 Tracking Background Functionality

Aquifer is designed to enforce security policy on user facing software. However, UI screens sometimes use background functionality such as daemons and file storage. When this occurs, Aquifer must carefully propagate policy labels between UI workflows.

**UI Screen Accessing a Daemon:** Daemons may be accessed by multiple workflows. Simply joining labels whenever a UI screen accesses a daemon will quickly result in all workflows having the same overly restrictive label. To avoid this, Aquifer uses intelligent tracking in daemons.

One method of intelligent tracking is to incorporate fine-grained tracking (e.g., Taint-Droid [Enc10] and CleanOS [Tan12]). Unfortunately, existing systems would require substantial retrofitting to enforce Aquifer policy. TaintDroid can only track 32 distinct identifiers. CleanOS extends TaintDroid to store identifiers in the taint tag bitvector; however, this storage cannot be used directly for Aquifer workflow labels. Furthermore, the source code for CleanOS was not available at the time of writing. Since the focus of this chapter is the UI
workflow security semantics, and not building another fine-grained data tracking framework, we reduced the scope of our tracking to OS-visible objects allowing coarse kernel mediation (i.e., files).

By restricting Aquifer to tracking files, we only need to track open file descriptors sent between UI screens and daemons. Android applications can pass file descriptors through binder. This commonly occurs with content provider components. Consider an activity component in application $A$ that wants to read an image file that is owned by application $B$. App $B$ can use a content provider component to share the image file with other applications without the image file being world readable. To do this, app $B$ allows app $A$ to query its content provider for a content URI, or passes the content URI directly to app $A$ (e.g., content://app_b/img/42). App $A$ can then open a `FileInputStream` for app $B$’s content provider using this URI. Behind the scenes, app $B$’s process will open the image file and pass the open file descriptor to app $A$ using binder. App $A$ can then read from the image file as if it opened the file itself.

Aquifer for Android implicitly tracks file descriptors in daemons by leveraging Linux’s `file_permission` LSM hook. This hook is invoked whenever an inode is read or written, as opposed to the commonly used `inode_permission` hook, which is invoked when the file is opened. `file_permission` provides Aquifer the file and the process performing the read or write, regardless of how the process obtained the file descriptor. Using `file_permission` also avoids ambiguous read-write file open masks, as well as properly propagating labels when the workflow label changes between file open and file write. However, these advantages come at the cost of degraded performance that results from retrieving the file’s label for each read and write operation.

**UI Screen Accessing a File:** By using `file_permission`, Aquifer leverages the Linux kernel’s tracking of file descriptors. Hence, even when a file is written through a daemon, Aquifer sees the UI screen accessing the file directly. When a process in a UI workflow reads or writes a file, Aquifer propagates the workflow label to and from the file in a standard way. Let the workflow have label $L_W$ and the file have label $L_F$. If the UI screen writes to a file, the file's label is updated to $L_W \sqcup L_F$. If the UI screen reads from a file, the UI workflow label becomes $L_W \sqcap L_F$.

To accomplish these updates, Aquifer relies on a kernel module and the userspace
Aquifer Service. When a file is read or written, a kernel hook extracts $L_F$ from the file (e.g., from its xattr) and notifies the Aquifer Service via an upcall, sending $L_F$ and the access mode. The Aquifer Service updates $L_W$ (if necessary) and returns a new $L_F$ (if necessary). The kernel module then stores the new $L_F$ with the file (e.g., in its xattr) if necessary.

Finally, propagating labels to persistent storage using file granularity means that Aquifer cannot handle sub-file data items such as database records. This limitation is currently in place for implementation and performance reasons.

5.5 Implementation

We implemented Aquifer for Android v4.0.3 (ICS) and the Linux Kernel v3.0.8 (omap). Aquifer adds approximately 2,200 lines of code in the Android Framework, and approximately 1,000 lines in the kernel. The source code is available at http://research.csc.ncsu.edu/security/aquifer.

The core userspace implementation is the Aquifer Service, a new system service responsible for maintaining the workflow abstraction and policy language calculus. The Aquifer Service is invoked by hooks placed in Android's ActivityManager service. These hooks inform Aquifer when system state changes affect the UI workflow state. The hooks are also used to filter intent resolution before presenting results to the user. The Aquifer Service also exposes an API to applications to safely add and modify their owner policies.

Aquifer uses a Linux security module (LSM) to mediate file access and a file descriptor transfer between processes. We use the file_permission LSM hook to only propagate the label if the data is read or written. The file policy is stored in extended attributes (xattrs), and the Aquifer LSM forwards file events and file policy to the Aquifer Service via a netlink socket. We also ensure that the SDcard is formatted to support xattrs.

The final component of our implementation is the Aquifer device driver, which provides a channel for the userspace Aquifer Service to communicate with the Aquifer LSM. The Aquifer Service uses this interface to manipulate the network access privilege of a process. The Aquifer Service also sets up the netlink socket with the LSM via this interface to receive events about file accesses.
5.6 Evaluation

We now evaluate Aquifer by assessing the need and appropriateness of its protection, proving the safety of label joins, and measuring the performance overhead. We also provide three case studies to demonstrate Aquifer in practice.

5.6.1 Application Survey

To understand the need for Aquifer and addressing the data intermediary problem, we performed a manual survey of Android applications.

Survey Setup: We selected the top 50 free Android applications from 10 categories in the Google Play Store (500 apps total). We chose categories based on use of privacy-sensitive application-specific data or the ability to use such data. For example, we omitted game-related categories, news and magazines, etc. We selected the following categories: Business, Communication, Media and Video, Music and Audio, Photography, Personalization, Productivity, Shopping, Social, and Tools.

Our application survey began by reading the market description of the application. For example, we identified if it creates or acquires data from the cloud. If we could identify a potential need for Aquifer, we studied the application manifest and manually ran the application as needed. Specifically, we looked at the types of interaction an application uses, e.g., complete isolation, data sharing in workflows, storing data in shared storage, as well as the type of data that was shared, i.e., we ignored data with no security or privacy value. Finally, we created a list of workflows that each app can be a part of to gain insight into how Aquifer’s policies could enhance application security.

Results: Table 5.1 provides the statistics from our study. We found a number of data sources that produced and shared data. Apps that did not produce any data, but processed data from other apps, were classified as intermediaries. We identified a larger number of intermediaries, which suggests more applications provide data services than produce data. This motivates the need to address the data intermediary problem. We also categorized applications based on the usefulness of Aquifer’s export and required restriction policies. These results motivate the appropriateness of Aquifer policy.

The application study also identified many interesting use cases. For example, some
applications facilitate business meetings by sharing of files during meetings. Aquifer can be used to help protect confidential business files against inadvertent exposure. We also identified many free applications that provide value-add capabilities, e.g., image transformation. There are reasons why users may wish to edit photographs on the phone. The user may wish to ensure the intermediary does not export copies, particularly if the user is a professional photographer.

### 5.6.2 Security Evaluation

Aquifer specifically seeks to protect application-specific data that cannot be enforced by system security policy. The security and privacy sensitivity of application-specific data is often only known to the developer and the user. We seek to reduce the onus on the user by having developers specify security policy. We note that app developers already participate in policy by specifying which permissions an app uses, and assigning permissions to restrict app interfaces.

Aquifer allows app developers to specify host export restrictions on data used by a UI workflow. The policy for a UI workflow is maintained in a workflow label $L$ (Definition 4). When information from one UI workflow is propagated to another UI workflow via files, Aquifer merges the two workflow labels using the join ($\cap$) operator (Definition 8). Section 5.3 claimed the join operation ensures the resulting label is at least as restrictive as both the original labels.

We formally prove the safety of the join operation and hence of the Aquifer policy language. We do this in two parts. First, we define an effective restriction relation that ensures the evaluated policy is more restrictive. Then, we define an owner restriction relation that ensures that all of an owner's restrictions are maintained. This is important, because while $L_2$ may be effectively more restrictive than $L_1$, an owner's restrictions may

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Number of Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data sources</td>
<td>85 (17%)</td>
</tr>
<tr>
<td>Data intermediaries</td>
<td>140 (28%)</td>
</tr>
<tr>
<td>Value from Export Policy</td>
<td>70 (14%)</td>
</tr>
<tr>
<td>Value from Regulate Policy</td>
<td>78 (15.6%)</td>
</tr>
</tbody>
</table>

---

Table 5.1 Application Survey Results
be changed at a later time by another owner such that $L_2$ is no longer more restrictive than $L_1$. With these two definitions, we can define an overall restriction relation that is needed to prove the safety of Aquifer. The formal proof is provided in Appendix A.

### 5.6.3 Performance Evaluation

To understand the performance overhead of Aquifer, we performed several microbenchmarks. The experiments were performed on a Samsung Galaxy Nexus (maguro) running Android v4.0.3 and Aquifer built on the same version. We performed each experiment at least 50 times. Average results with 95% confidence intervals are shown in Table 5.2.

#### App load time:
Aquifer initializes its UI workflow structures when the first application is loaded. This consists of creating a new label and data structures for $W_V$ and $W_R$ to maintain the workflow state. We compared the time to start the first application of a UI workflow in Aquifer to a baseline application load time in Android. The average overhead is 3.58 ms, which is negligible.

#### App filtering:
Aquifer filters the potential target applications when Android uses an implicit intent to start an activity component. We measured the time between sending an intent message and the resolution of the final list of applications presented to the user. Aquifer only causes a negligible delay of 1.1 ms.

#### Network access check:
Aquifer places a hook in the kernel that is called every time a process attempts to access the network. For this experiment, we created an application with an activity component that attempts to access the network repeatedly. Since Android already performs a similar check to enforce its INTERNET permission, Aquifer’s additional checks have negligible impact.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Android</th>
<th>Aquifer</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>App load</td>
<td>188.49±5.36 ms</td>
<td>192.07±6.30 ms</td>
<td>1.9%</td>
</tr>
<tr>
<td>App filter</td>
<td>194.12±7.91 ms</td>
<td>195.22±7.52 ms</td>
<td>0.55%</td>
</tr>
<tr>
<td>Net access</td>
<td>108.60±6.48 ms</td>
<td>109.64±6.31 ms</td>
<td>0.53%</td>
</tr>
<tr>
<td>Policy change</td>
<td>-</td>
<td>1.98±1.27 ms</td>
<td>-</td>
</tr>
<tr>
<td>File Read (1MB)</td>
<td>4.76±0.09 ms</td>
<td>5.23±0.22 ms</td>
<td>9.87%</td>
</tr>
<tr>
<td>File Write (1MB)</td>
<td>23.89±0.45 ms</td>
<td>25.44±0.86 ms</td>
<td>6.49%</td>
</tr>
</tbody>
</table>
Workflow policy change: An application with an active activity can modify the UI workflow policy label, which requires recalculation of the effective policy and reassignment of network privileges to all workflow participants. This policy re-evaluation only takes 1.98 ms.

Label propagation on read and write: Each file read operation requires Aquifer to retrieve the file’s label from its xattr and join it to the workflow’s label. Each file write operation requires Aquifer to retrieve the file’s label, modify it, and store the new label. For this experiment, we measured the overhead of reading and writing a 1MB file with a small workflow policy. We performed each read and write 50 times, flushing after each write, and sleeping 500 ms between consecutive measurements. Table 5.2 shows an overhead of 6.49% for writes and 9.87% for reads. Note that while Aquifer writes are more complex than reads, the read overhead is greater, because the read time is significantly less than the write time. Furthermore, a production version of Aquifer could cache policies in memory to avoid unnecessary xattr operations.

To further investigate the read and write overhead, we performed a more detailed study of the time required. We repeated the previous experiment, but used a range of workflow label sizes and complexities. We started with a simple single owner label containing an owner policy of 148 bytes and increased gradually to a fairly complex label containing multiple owners and occupying 1KB.

Figure 5.4 shows the time required for Aquifer to perform the read and write label propagation based on the policy size. The horizontal line shows the time to perform the
AquiferList exportList = new AquiferList();
exportList.add(this.getPackageName());

AquiferFilter filter = new AquiferFilter();
filter.addTarget(android.intent.ACTION_SEND, this.getPackageName());

AquiferPolicy policy = new AquiferPolicy();
policy.setExportList(exportList);
policy.setFilter(filter);

IAquiferService aquifer = IAquiferService.Stub.asInterface(ServiceManager.getService("Aquifer"));
aquifer.addPolicy(policy);

Figure 5.5 Aquifer policy modifications to K-9E Mail

read and write in Android without Aquifer modifications. There are four contributors to this overhead: 1) context switches when transporting labels from kernel space to user space and vice versa; 2) performing the xattr operations, 3) marshalling and unmarshalling the policy to and from the binary form; and 4) copying the data itself.

Figure 5.4 shows a relatively constant overhead, indicating that the setup cost of context switches and xattr operations overwhelms the cost of marshalling data and copying data between buffers. Finally, the overhead for reading and writing empty labels is negligible, as we avoid propagating empty labels.

5.6.4 Case Studies

To demonstrate how Aquifer works in practice, we performed three case studies involving open source Android applications such as K-9 Mail, OI File Manager, and PDFView.

5.6.4.1 Case Study 1 (Confidential PDF)

K-9 Mail is an open source fork of the original Email client in the Android Open Source Project (AOSP). We modified K-9 Mail to create K-9E Mail, an enterprise email client for use by the employees of a fictional enterprise. We also used the open source PDFView application, which we modified to emulate an intermediary that backs up the files accessed by the user to the user’s account in the cloud.

Our modifications of PDFView include 1) sending the PDF file to a network server, and
2) saving a version of a PDF file, and then on a later invocation of PDFView, opening the saved file and sending it to the network. PDFView does not go out of its way to collect data, rather data is collected only as a consequence of using it.

K-9E Mail allows the user to view attachments in other applications. For our case study, we use an Email with the file contract.pdf attached. When the user selects to view contract.pdf, K-9E Mail creates an intent message with the implicit address ACTION-_VIEW and the datatype set to application/pdf. When K-9E Mail uses this intent to start an activity, Android displays a chooser allowing the user to select the viewer. In our case study, this chooser contains the default DocumentViewer app and our modified PDFView app. We verified that the PDF could be viewed by both DocumentViewer and PDFView while running in the Aquifer enhanced Android framework, without any modification to either app. When we viewed contract.pdf, PDFView successfully exported the PDF as designed.

We then modified K-9E Mail to be Aquifer-aware. For the case study, we included logic to identify a PDF as confidential if the filename contains strings such as “contract,” “confidential,” “secret,” etc. Note that we used this classification scheme purely for demonstration purposes. A production version of an Aquifer-aware Email client could be much more intelligent (e.g., scan the subject and body for keywords, use predefined X-Headers, etc.). The Email client should also provide the user visual clues that the attachment is treated as confidential, and potentially a method to declassify an attachment in the event of false labeling.

Our second modification was to set the owner policy for the UI workflow before a confidential attachment is viewed. We used the following owner policy.

\[
E = \{\text{K9EMail}\} \\
R = \{} \\
F = \{(\text{ACTION_SEND}, \{\text{K9EMail}\})\}
\]

This policy ensures that only K-9E Mail can export the data, and if any application in the UI workflow uses the ACTION_SEND action string to start an activity, only K-9E Mail will be displayed, filtering out other options (e.g., YahooMail, Hotmail), as shown in Figure 5.6. Adding this policy to K-9 Mail required very few changes, as shown in Figure 5.5.
We then re-performed our previous experiment. This time, when PDFView attempted to send `contract.pdf`, it could not reach the network. Furthermore, when PDFView saved a copy of `contract.pdf`, the workflow label was copied with it. When we later invoked PDFView as part of an unrestricted UI workflow, it read `contract.pdf` (due to our changes) and the workflow was successfully labeled, again keeping PDFView from exporting the document.

5.6.4.2 Case Study 2 (Choosers)

The previous case study shows how K-9E Mail can share data while ensuring that only it can export the data off the device. In this case study, we demonstrate how K-9E Mail can allow a larger set of applications to export the data only if the user’s consent is provided.

For this case study, K-9E Mail trusts all other applications to send confidential docu-
ments off the host, but only if the user selects the file as part of a workflow. This policy is valuable to prevent accidental backup to cloud storage by other applications the user might have installed. This policy goal is accomplished using a trusted chooser application and a \textit{require} restriction. For example, if K-9E Mail trusts the OI File Manager, the following policy can protect documents saved to the SDcard from accidental disclosure.

\begin{align*}
E &= \{\text{ALL}\} \\
R &= \{\text{OI File Manager}\} \\
F &= \{\} 
\end{align*}

Using this policy, Aquifer allows the original K-9 Mail app to send the saved attached document when both, 1) starting the OI File Manager from K-9 Mail to choose an attachment, and 2) starting OI File Manager first and sharing the document with K-9 Mail.

5.6.4.3 Case Study 3 (Document Viewers)

Our final case study evaluates whether or not Aquifer policies are compatible with popular data intermediaries. We downloaded 25 of the most popular free document and image viewers and editors. Each was shared a file that has an Aquifer policy that prevents the intermediary from opening network connections. For the 25 applications, we encountered 0 application crashes due to access control failures. We found that seven of the applications (e.g., KingSoft Office, Olive office) contain advertisement libraries that immediately make network connections, before displaying the document. However, when Aquifer denies these network connections, the applications handle the denied connection without error and without usability impact (except for the absence of the ad). This use case supports our hypothesis that many data intermediary applications are built with modularity in mind and have limited dependencies on the Internet.

5.7 Discussion

Aquifer policy specification may lead to usability failures if application developers do not predict all of the ways in which the user might construct a UI workflow. One potential case
is when regulate restrictions can conflict with filters. Regulate restrictions require an app to participate on a workflow. However, if that app is not included in a workflow filter, the user may never be able to navigate through it. This example demonstrates a need for developers to coordinate on Aquifer policy at some level.

Another type of unexpected usability failure due to Aquifer policy results when a user clicks on a hyperlink in a protected document. If the Web browser is not in the export list, it will fail to navigate to the URL when launched from the workflow containing the document. Technically, the URL was part of the document and should not be exported. However, a policy may wish to include a trusted Web browser in the export list to ensure hyperlink functionality.

Finally, as discussed in our first case study, there are various situations when the app developer may need to indicate to the user that data is classified in order to avoid user confusion that may lead to access control violations. Such situations must be addressed on an application-specific basis.

## 5.8 Summary

Modern operating systems have changed both the way users use software and the underlying security architecture. These two changes make accidental data disclosures easier. To address this problem, we presented the Aquifer security framework that assigns host export restrictions on all data accessed as part of a UI workflow. Our key insight was that when applications in modern operating systems share data, it is part of a larger workflow to perform a user task. Each application on the UI workflow is a potential data owner, and therefore can contribute to the security restrictions. The restrictions are retained with data as it is written to storage and propagated to future UI workflows that read it. In doing so, we enable applications to sensibly retain control of their data after it has been shared as part of the user's tasks.
Application-based modern operating systems, such as Android, thrive on their rich application ecosystems. Applications integrate with each other to perform complex user tasks, providing a seamless user experience. To work together, applications often share user data with one another. Such sharing exposes the user’s private and enterprise information to the risk of exfiltration from the device. For example, a confidential email attachment opened in a third party document editor (e.g., Microsoft Word) could be exported from the device if the editor is malicious or compromised.

Android’s permission framework is used to protect application data, and applications are security principals that own the data they bring to the device. However, permissions are only enforced at the first point of access. Once data has been copied to the memory of an untrusted third party application’s process, through user-directed sharing or explicit
permission to access storage, neither the OS nor the data owner application can enforce any restrictions on its use. The untrusted application may then export the data to a remote adversary. This problem is generic to operating systems that provide only data protection, but not data secrecy, and can be solved by integrating information flow secrecy guarantees.

Classic information flow control (IFC) [Den76] only captures well-known data types through a centralized policy. On Android, data is oftentimes not well-known or is application-specific (e.g., email attachments, notes). Therefore, Android needs decentralized IFC (DIFC) which allows data owners (i.e., applications) to specify the policy for their own data types. Many DIFC models have been proposed for traditional operating environments [ML97; Van07; Zel06; Kro07] as well as Android [NE13; Jia13; XW15].

However, existing DIFC solutions for Android cannot achieve both security and practicality. For instance, an Android application's components (i.e., activities, services, content providers and broadcast receivers) are instantiated in the same application process by default. These components are often involved in separate user tasks for which they may process data from different sources and secrecy contexts. That is, Android's components share state from different calls in the application process's memory. This shared state makes enforcing DIFC on the process hard, as the combined restrictions from all secrecy contexts would make individual components unusable. To solve this problem, prior Android DIFC proposals make the application execute only one user task at a time, depriving the user of multi-tasking capabilities. The two ways in which such single task behavior is forced are also detrimental to backwards compatibility: 1) killing application processes per new call, which could result in dangling state, and 2) blocking until the application voluntarily exits, which could lead to deadlocks. DIFC enforcement on Android requires a more practical solution for separating shared state in memory.

Similarly, Android apps also share state on storage. An application's components have common access rights to the application-specific internal storage, while nearly all applications have permission protected access to the external storage (i.e., SD card). IFC restrictions propagated to storage with the data flow may affect the availability of common files used by all secrecy contexts (e.g., application settings). While some Android DIFC proposals attempt to separate this shared state on storage, they fail to do so transparently; i.e., without denying access to application resources and without requiring applications to be modified. To summarize, prior DIFC systems for Android do not separate shared state
in memory and on storage with transparency and backwards compatibility.

In this paper, we present Weir,\(^1\) a practical DIFC system for Android. Weir allows data owner applications to set secrecy policies and control the export of their data to the network. Apart from the data owners, and applications that want to explicitly use Weir to change their labels, all other applications can execute unmodified. Weir solves the problem of shared state by separating memory and storage for different secrecy contexts through polyinstantiation. That is, Weir creates and manages instances of the application, its components, and its storage for each secrecy context that the application is called from, providing availability along with context-sensitive separation. Our model is transparent to applications; i.e., applications that do not use Weir may execute oblivious to Weir's enforcement of secrecy contexts.

We term our approach as “lazy” polyinstantiation, as Weir creates a new instance of a resource only if needed, i.e., if there is no existing instance whose secrecy context matches the caller's. Additionally, Weir provides the novel primitive of domain declassification for practical and secure declassification in Android's network-driven environment. Our approach allows data owners to articulate trust in the receiver of data (i.e., trusted network domain).

This chapter makes the following contributions:

- **We identify the challenges of integrating DIFC into Android.** Using these challenges, we then derive the goals for designing DIFC enforcement for Android.

- **We introduce the mechanism of “lazy” polyinstantiation** for context-sensitive separation of the shared state. Further, we provide the primitive of Domain Declassification for practical declassification in Android's network-driven environment.

- **We design and implement Weir on Android.** Weir incurs less than 4ms overhead for starting components. Weir's design ensures backwards compatibility. We demonstrate Weir's utility with a case study using the K-9 Mail application.

While Weir presents a mechanism that is independent of the actual policy syntax, our implementation uses the policy syntax of the Flume DIFC model \([\text{Kro07}]\). Weir extends Flume by allowing implicit label propagation, i.e., floating labels, for backwards compatibility with unmodified applications. Since floating labels are by themselves susceptible

\(^1\)Weir: A small dam that alters the flow of a river.
to high bandwidth information leaks [Den76], we show how Weir's use of floating labels is inherently resistant to such leaks. Note that while language-level IFC models [Ste14; Ste11b; Ste15] often incorporate checks that prevent implicit flows due to floating labels, our solution addresses the challenges faced by OS-level floating label DIFC systems [Van07; Jia13]. Finally, we note that Weir provides practical DIFC enforcement semantics for Android, and the usability aspect of DIFC policy and enforcement will be explored in future work.

6.1 Motivation for Practical and Secure Data Sharing

In this section, we motivate the requirement of information flow secrecy guarantees on Android with an example.

6.1.1 Motivating Example and Threat Model

Consider Alice, an enterprise user in a BYOD (bring your own device) context. Alice receives an email in the enterprise OfficeEmail application with an attached report. She edits the report in a document editor, WPS Office, and saves a copy on the SD card, accessible to all applications that have the READ_EXTERNAL_STORAGE permission. Later, Alice uses the ES File Explorer to browse for the report, edits it in WPS Office, and then shares it with OfficeEmail to reply to the initial email.

To perform their functions, untrusted third party data managers such as ES File Explorer require broad storage access. Even without direct access, user-initiated data sharing grants data editors like WPS Office access to confidential data. If ES File Explorer or WPS Office were malicious or compromised, they could export Alice's confidential data to an adversary's remote server.

Threat Model and Assumptions: In this chapter, we seek to enable legitimate use of third party applications to process secret user data, while providing data security and preventing unauthorized data disclosure. Our goal is to protect against both malicious as well as accidental disclosure of data by third party applications. For this purpose, our solution, Weir, must mediate access to the network. To prevent application collusion and confused deputy attacks, Weir must mediate and track explicit data flows among applications. Weir
must also track the flow of secret data to the file system. Unlike fine-grained taint tracking approaches [Enc10], Weir must defend against implicit data flows in application code.

Weir's trusted computing base (TCB) consists of the Android OS (i.e., kernel and system services), and core network services (e.g., DNS). Weir assumes a non-rooted device, as an adversary with superuser privileges may compromise OS integrity. Further, we assume correct policy specification by the data owner applications, specifically regarding declassification. To prevent timing and covert channels based on shared hardware resources (e.g., a hardware cache), the only alternative is denying data access to secret data or the shared resource. Weir does not defend against such channels, which are notoriously hard to prevent in DIFC OSes in general.

6.1.2 Why Information Flow Control (IFC)?

Android uses its permission framework to protect user data. While permissions provide protection at the first point of access, the protection is not extended to data copied over to the intermediary application (e.g., WPS Office). The user or the data owner application (e.g., OfficeEmail) have no control over the flow of data once it is shared. Thus, unauthorized data disclosure is an information flow problem that Android’s permissions are not designed to solve. Further, coarse-grained containers (e.g., Samsung Knox [Sam13], Android for Work [Anda]) cannot be used for secrecy as they do not address threats from within the container, specifically the accidental export of secret data by a trusted application or the potential compromise of a trusted application within the container.

IFC [Den76] provides definition and enforcement of the allowable data flows in the system, as described in Chapter 3. Hence, secrecy guarantees can be leveraged through IFC in order to prevent unauthorized export of sensitive user data. A centralized IFC policy can only describe the secrecy constraints for well known data objects (e.g., location, IMEI). Therefore, we use decentralized IFC (DIFC) [ML97], to protect dynamic and previously unknown application-specific data, such as Alice’s secret report received by OfficeEmail.
6.2 DIFC Challenges on Android

We now discuss the aspect that make practical and secure DIFC enforcement challenging on Android, i.e., 1) Android’s default multitasking, 2) background components, 3) shared application storage and 4) Internet-driven environment. Further, we describe how previous DIFC systems for Android fail to address these challenges.

6.2.1 Multitasking on Android

Android’s user interface is organized into user tasks that represent the user’s high level objectives. By default, Android is designed for multitasking, and an application can be instantiated in multiple simultaneous tasks through its activities. Additionally, a “standard” (default) activity can be instantiated multiple times in one or more simultaneous tasks [Andb].

Consider the two tasks in Figure 6.1. In Task 1, the user opens a secret PDF (e.g., a contract) in WPS Office, which loads in its PDF Activity, and then shares it with the Evernote app. In Task 2, the user opens a non-confidential PDF (e.g., a published paper) in another instance of WPS Office’s PDF Activity. When the user chooses to print the PDF, the instance of PDF Activity in Task 2 sends it to WPS Office’s internal Print Activity component. As seen in Figure 6.1 multiple activities from the WPS Office app, as well as multiple instances of the same activity (i.e., PDF Activity) run in the same process, and hence share data in memory (e.g., through global variables).

As the two instances of the PDF Activity are instantiated with data of different secrecy
requirements (i.e., secret.pdf and public.pdf), we can say that they run in different secrecy contexts. Enforcing the DIFC policy on the process due to the sensitive nature of Task 1’s data would also unjustifiably restrict the non-sensitive Task 2. A simple approach of forcing every component to start in a new process, may break application components that share data via global variables initialized in the same process. For example, the internal Print Activity may try to access a global variable initialized by the PDF Activity, but may crash if the former is forced to start in a separate process. To summarize, component instances performing different user tasks often share state in process memory, making process-level DIFC enforcement challenging.

6.2.2 Background components

As described in Section 2.2, Android’s services and content providers are background components. Further, only a single instance of a service or a content provider is shared among all of an application's instances. Moreover, services and content providers can also communicate with other applications. As a result, different secrecy contexts may mix in the memory of a single background component instance.

To elaborate, as a single background component may communicate with component instances that may be running in various secrecy contexts, explicit labeling of background components is difficult. Moreover, if floating labels (described in Section 2.3) are applied, then the provider may accumulate the labels of all the secrecy contexts it communicates with, and then propagate the its new label back to the components that are connected to it. This results in what is termed as label explosion, where the entire system acquires a large, restrictive label that cannot be declassified by any single security principal. Note that background components may run in the shared application process by default, and a naive solution of restarting a background component's process for each new call is infeasible, as it would crash the other components (e.g., a foreground activity) running in that process.

6.2.3 Public and Private Application storage

Android provides each application with its own private storage shared amongst all of the application's runtime instances, irrespective of their secrecy context. For example, both the
sensitive and non-sensitive instances of WPS Office’s PDF Activity access the same user settings stored in the application's private directory.

Since storage must be available in every possible secrecy context, files must use floating labels. This may then result in a label explosion due to propagation of sensitive secrecy labels to files shared by all instances of the application. Label explosion to storage would be persistent, spanning reboots, as long as labeled data is present on storage. Additionally, the availability of public storage (i.e., the SD card) shared by all applications with the required permissions increases the possibility and impact of label explosion. Thus, separating this shared state on storage without causing applications to lose access is a challenge for practical DIFC enforcement.

6.2.4 Internet-driven environment

Android applications are often connected to the Internet. In such an environment with frequent network export, explicit declassification by the data owner is inefficient. Delegation of the declassification privilege to allow export without the owner’s intervention would bloat the application's TCB. Additionally, existing declassification mechanisms described in Chapter 2 make the policy decision based on the identity of the security principal performing the export. On Android, such mechanisms would limit the user to using a small subset of applications for data export (i.e., out of the 2 million applications on Google Play [Sta]), which would be detrimental to adoption of DIFC on Android.

6.2.5 Prior DIFC Proposals for Android

We discuss three prior DIFC proposals for Android, namely Aquifer [NE13], Jia et al. [Jia13] and Maxoid [XW15], all of which are OS-level DIFC systems. Our objective is to understand the design choices made by these systems, with respect to the challenges described previously.

1. Aquifer: Our prior work, Aquifer [NE13], provides protection against accidental data disclosure, by tracking the flow of data through applications, and enforcing the declassification policy for network export.

   For seamless data sharing between applications, Aquifer uses the floating labels de-
scribed in Section 2.3. To limit label explosion, Aquifer does not label background components, and hence can only prevent accidental data disclosure. On the other hand, Aquifer labels storage, but does not claim to mitigate label explosion on storage. Further, to prevent different secrecy requirements for data in the memory of a single process, Aquifer disables Android’s multi-tasking and restarts the process of the existing instance when the application is called from another secrecy context. Finally, Aquifer’s declassification policy allows the data owner to explicitly specify the security principal that may export data, or a condition on the call chain for implicit export.

2. Jia et al.: The DIFC system by Jia et al. [Jia13] also uses floating labels to support general-purpose applications, but supports strict secrecy policies (i.e., relative to Aquifer) that may restrict data sharing among applications if needed.

Contrary to Aquifer, the system propagates labels to background components, providing stronger protection against malicious data exfiltration. At the same time, the system makes no claims of controlling label explosion via background components or storage. The system uses Flume’s capabilities [Kro07] for declassification. This work also acknowledges the challenge of multi-tasking along with DIFC enforcement, and disallows multi-tasking by blocking new calls to an application until all of its components voluntarily exit. Since Android components do not exit by themselves like conventional OS programs, such blocking could potentially lead to deadlocks.

3. Maxoid: Xu and Witchel [XW15] provide an alternate approach to file system labeling to prevent label explosion in Maxoid, by using file system polyinstantiation [LS05] to separate differently labeled data on disk.

Maxoid addresses new calls to existing labeled instances in a manner similar to Aquifer’s; i.e., by restarting the instance. Additionally, Maxoid prevents access to background components from labeled instances, thereby preventing label explosion, although at the cost of backwards compatibility. On the other hand, Maxoid considers overt data flows through Binder IPC as declassification, unlike the system by Jia et al. that mediates such communication. Finally, Maxoid modifies system content providers (e.g., Contacts) to use a SQL proxy, in order to extend its label separation into system content providers. As a result, Maxoid’s storage separation is unavailable for use by content providers in unmodified third party applications.
**Takeaways:** Prior approaches demonstrate the possibility of DIFC on Android, and make convincing arguments in favor of using floating labels, mainly for backwards compatibility with Android's unpredictable data flows. At the same time, we observe that in prior systems it becomes necessary to relax either security or backwards compatibility in order to use floating labels on Android (e.g., with background components). Additionally, prior approaches recognize the need to separate different secrecy contexts in process memory, but the proposed solutions disable Android's default multi-tasking. Finally, in systems that aim to address label explosion on storage, only separating the shared state on storage without addressing the shared state in memory may be insufficient to support unmodified applications.

### 6.3 Design Goals for Practical Enforcement

Existing DIFC approaches for Android do not fulfill the challenges described in Section 6.2. Our objective is to design a DIFC mechanism for Android that provides secrecy guarantees while maintaining backwards compatibility with the Android application model. We seek a design where the only applications that have to be modified are the data owners that create and set labels, and applications that want to explicitly change their labels. Our design goals are as follows:

**G1** *Separation of shared state in memory.* DIFC enforcement must ensure that data from different secrecy contexts is always separated in memory, preferably in the memory of different processes. Process-level enforcement can then be used to mediate flows between differently labeled data.

**G2** *Separation of shared state on storage.* DIFC enforcement must ensure that data from different secrecy contexts is separate on persistent storage. For mediation by the OS, the separation must be at the level of OS objects (e.g., files, blocks).

**G3** *Transparency.* A naive implementation of goals **G1** and **G2** would affect the availability of components and storage. Our system must be transparent, i.e., applications that do not use the DIFC system must be able to operate oblivious to the enforcement.

---

2Killing existing processes or blocking can result in the killing of unrelated components sharing the process, or deadlocks, respectively.
**G4 Secure and Practical Declassification.** A DIFC system on Android should provide a declassification primitive that is both feasible (i.e., does not hinder the use of applications) and secure.

### 6.4 Weir Overview

In this chapter, we propose *Weir*, a practical and secure DIFC system for Android. *Weir*’s design is guided by the goals described in Section 6.3. We present an overview of *Weir* and its approach of *lazy polyinstantiation*, following which we describe individual aspects of our design.

*Weir* allows applications to define the policy for their data by creating their own security classes. *Weir*’s mechanism labels files (as objects) and processes (as subjects), granting the kernel complete mediation over all data flows among subjects and between subjects and objects. As *Weir*’s contributions are in its policy-independent mechanism, we only use the generic information flow terminology described in Section 2.3 for discussions involving the policy, and describe the actual policy structure used by our implementation in Section 6.6.

We use “floating labels” (described in Section 2.3), as explicit labels are hard to assign a priori in Android, where data flows are often user-directed and unpredictable. However, a naive use of floating labels cause certain components to acquire more labels due to involvement in multiple secrecy contexts, and eventually become unusable. This is because the logic of propagating floating labels is not context sensitive. We use polyinstantiation to make floating labels context sensitive, and hence separate the shared state in different
secrecy contexts in memory (G1) and on storage (G2). Our approach is in principal similar to context sensitive inter-procedural analysis that adds precision by considering the calling context when analyzing the target of a function call (e.g., summary functions and call strings [SP81], k-CFA [Shi91], and CFL-reachability [Rep98]). To our knowledge, context sensitivity has not been exploited in the scenario investigated in this chapter.

We describe polyinstantiation relative to explicit and floating labels with the example scenario in Figure 6.2, where an instance of component P with label \( \{ L_P \} \) tries to send a message to an instance of component Q with a label \( \{ L_Q \} \), and where \( \{ L_P \} \neq \{ L_Q \} \). In World A where only explicit labels are allowed, the message would be denied as Q would not be able to explicitly change its label to \( \{ L_P \} \) without a priori knowledge of P’s intention to send a message. In World B with floating labels, the flow would be automatically allowed, with Q’s new label implicitly set to a join of the two labels. While World B allows seamless communication, it does not prevent the two secrecy contexts (i.e., \( \{ L_P \} \) and \( \{ L_Q \} \)) from mixing, leading to the challenges we explored in Section 6.2. In World C, we use polyinstantiation along with floating labels, and a new instance of Q denoted as \( Q^1 \) is created in the caller’s context (i.e. with the caller’s label \( \{ L_P \} \)), separate from the original instance of Q with label \( \{ L_Q \} \). Thus, our approach allows the call to take place, without the mixing of secrecy contexts. The “lazy” aspect of our approach (not represented in the figure) is that we would reuse a previously created instance of Q, denoted \( Q^{past} \), if its label matched the caller’s label (i.e., \( \{ L_P \} \)). Additionally, while the new instance has an empty label (i.e., \( \{ \} \)) as the base (compile-time) label in our prototype, our model can be adapted to support a different base label.

Weir uses lazy polyinstantiation for all indirect inter-component calls (e.g., starting an activity, querying a content provider) (described in Section 2.2). Weir polyinstantiates processes, Android components and the file system, creating new instances of each for different secrecy contexts. Floating labels allow legacy apps to integrate into Weir without modification for making or receiving calls, while polyinstantiation adds context sensitivity. Weir’s use of floating labels supports process-level labeling along with application multi-tasking (G3), a more practical solution than the alternatives of killing existing instances [NE13; XW15] or indefinite blocking [Jia13].
6.5 Weir Design

In this section, we describe Weir’s component and process polyinstantiation, followed by the polyinstantiation of storage. We then discuss explicit label changes by existing component (and process) instances, the resultant challenges, and how Weir’s design resolves the challenges. Finally, we describe how Weir satisfies goal G4 through its domain declassification.

6.5.1 Lazy Component and Process Polyinstantiation

In order to satisfy goal G1, Weir must ensure that no two component instances with mismatching labels execute in the same process. At the same time, Weir must also make components available if the underlying Android enforcement (i.e., component permissions) permits. Therefore, Weir’s approach is to polyinstantiate components, so as to make them simultaneously available in processes with different secrecy labels, as separate instances running in those processes.

We initially explored an approach based on Android’s “multi-process” manifest attribute, designed to instantiate activity components in the caller’s process. Though this approach would work in some cases, forcing the “multi-process” attribute for all activities is not practical if the caller is from another application. In such cases, the target component runs with the caller application’s Linux UID and permissions, and loses all access to its internal app resources (i.e., private storage, internal app components). As a result, this option is only suitable for activities that can operate independent of their app’s resources, and would break apps if applied without the developer’s cooperation. Further, the caller’s application is exposed to the security risk of running untrusted code in its context. Thus, its security and backwards compatibility limitations make this approach impractical.

Instead, Weir’s approach refrains from modifying or affecting developer configurations. Weir polyinstantiates components within the application’s own context. Weir accounts for the process assignments made for components by the app developer, through the “android:process” manifest attribute (represented internally as the component’s process-Name). When assigning processes to newly created component instances, Weir ensures that components that were meant to run together (i.e., assigned the same processName),
still run together, but in an instance of the process with a matching label. We now describe our approach, followed by an example.

**Our approach:** As discussed previously, *Weir* uses floating labels for calls to application components, i.e., the callee component gets the caller’s label. On every new call, *Weir* retrieves the label of the caller (i.e., the `callerLabel`). *Weir* then checks if an instance of the desired component is running in a process whose label matches `callerLabel`. If one is found, the call is delivered to the matching component instance. If not, *Weir* creates a new instance of the called component.

When the target component instance is assigned, *Weir* must find a process to execute it. If the process associated with this component (i.e., `processName`) has a different label, *Weir* cannot execute the new instance in it, and has two options: 1) assign a polyinstantiated process that is associated with `processName` and has the label `callerLabel` or 2) create a new process associated with the `processName` with label `callerLabel`. As it is evident based on the first option, *Weir* maintains the association between a process (`processName`) and all its new instances created for various secrecy labels. This knowledge helps *Weir* in two ways. First, *Weir* can reuse such polyinstantiated processes, when a component must be instantiated with a label for which a polyinstantiated process already exists (i.e., “lazy” polyinstantiation). Second, the association ensures that the developer’s process allocation is adhered to; i.e., polyinstantiated components will not run in any application process, but only in the process associated with their `processName` and with a matching label; ensuring that components that are meant to run together do so.

If a process associated with the original `processName`, and whose label is `callerLabel` is not available, *Weir* creates a new process, whose name is formed by appending an index to the end of the original `processName`; the index increments every time the original `processName` is polyinstantiated.

To summarize, *Weir* creates new component instances and processes to handle calls from new secrecy labels. At the same time, *Weir* keeps track of all instances created in the past, and reuses existing component instances and processes if the labels match. A detailed description of the implementation aspects of our component and process assignment logic in Android’s Activity Manager service can be found in Appendix B.1. We now demonstrate our approach with an example.
Example: Consider an app with three components, activities A and B, and a service C. The developer sets the `processName` for A and B to be “procActivity”, whereas the `processName` for C is set to “procService”. This means that A and B are expected to run in the same process, while C runs in a separate process. Additionally, A is an exported activity, i.e., it can be called by other apps. C is an exported service that can be bound to using the `bindService` call. The app is programmed such that when A is started, it starts B, following which B starts C. Using Figure 6.3, we demonstrate Weir’s lazy polyinstantiation of A, B, and C and the processes “procActivity” and “procService”.

In Step 1 (Figure 6.3a), A is first called by an unlabeled caller; i.e., the `callerLabel` is empty. A new instance of A is created, and a new process by the name “procActivity” is started for it. Then, A calls B. The label of A’s process is empty, so B is also instantiated with an empty label, in the matching process, i.e., “procActivity”. B then calls C, which is instantiated in the new process “procService”.

In the Step 2 (Figure 6.3b), A is called from a caller with `callerLabel = {L_1}`. Weir
cannot deliver the call to the existing instance of $A$, as its process has a mismatching label (i.e., $\text{callerLabel} = \{L_1\} \neq \{\}\) and allowing it to process the current call would violate goal $G_1$. Thus, $\text{Weir}$ creates a new instance of $A$ for this call. As there are no processes associated with “procActivity” and with the label $\{L_1\}$, $\text{Weir}$ also allocates a new process “procActivity_0” to host this instance. Thus, for this call, a new instance of $A$ is started in a new process “procActivity_0”, whose label is set to $\{L_1\}$. When this instance of $A$ starts $B$, the call is treated as a call to $B$ with $\text{callerLabel} = \{L_1\}$, the caller being $A$’s new instance with label $\{L_1\}$. Since $\text{Weir}$ keeps records of all the new processes created for polyinstantiation, it starts a new instance of $B$ in the process that is associated with $B$’s original process “procActivity”, and has a matching label $\{L_1\}$, i.e., “procActivity_0”. Reusing an existing process with a matching label is an example of “lazy” polyinstantiation. When this instance of $B$ starts $C$, $\text{Weir}$ creates a new instance of $C$’s due to mismatching labels, and that instance is started in a new process “procService_0” with the label $\{L_1\}$.

In Step 3 (Figure 6.3c), $\text{bindService}$ is called on $C$ with the label $\text{callerLabel} = \{L_2\}$. Since the caller’s label $\{L_2\}$ mismatches with the two existing instances of $C$ that are running with labels $\{\}$ and $\{L_1\}$, a new instance of $C$ is created. As there are no processes associated with “procService” that have a label matching $\{L_2\}$, a new process “procService_1” is created to host the new instance. Note that all of these instances and processes exist simultaneously, as shown in the figures. If $C$ is called again with the label $\text{callerLabel} = \{L_2\}$, $\text{Weir}$ does not have to create a new instance, and the call is delivered to the existing instance of $C$ running in process with a matching label, i.e., “procService_1”. This is another instance of the “lazy” aspect of polyinstantiation, as $\text{Weir}$ reuses existing component and process instances for a call from the same secrecy context.

In the example, we see that $\text{Weir}$ polyinstantiates components and processes for every call for which matching labeled instances do not exist, ensuring separation in memory. Further, $\text{Weir}$’s approach of “lazy polyinstantiation” ensures that polyinstantiation only occurs when the caller has a label that does not have a matching component instance, process or both. We also observe that components configured by the developer to run together (i.e., in the same process) still run together; i.e., our approach is transparent to the application, satisfying Goal $G_3$. For example, $A$ and $B$ exist together in both the unlabeled as well as labeled processes. This ensures that access to global variables defined in the same process’s memory would not break, as it would by force-starting every component in
its own separate process. To summarize, Weir polyinstantiates components into different processes, separating application instances along different secrecy contexts. We discuss the security of polyinstantiation in Section 6.7.

Our approach supports polyinstantiation of all Android component types declared in the application manifest, i.e., activities, services, content providers and broadcast receivers. An exception is broadcast receivers registered at runtime, which are instantiated at registration in the secrecy context of the registering process, and hence not subject to further polyinstantiation. Any future broadcasts to such receivers are treated as direct calls subject to strict DIFC label checks (similar to Binder calls). The back end of the content providers, i.e., databases stored on the app's internal or external storage, are managed using the polyinstantiation of internal and external app storage, as we describe in Section 6.5.2.

6.5.2 Polyinstantiation of Application Storage

The previous section demonstrates Weir’s use of lazy polyinstantiation to separate secrecy contexts in memory. To prevent restrictive labeling of shared storage by processes running in sensitive contexts, Weir extends this separation to the storage as well, satisfying goal G2. Further, Weir achieves this separation without disallowing storage access to instances in sensitive secrecy contexts, satisfying goal G3. Similar to Section 6.5.1, we first describe our approach, followed by an example. We then describe how Weir labels the file system to protect against malicious applications that purposely violate the separation to access data of higher secrecy.

Our approach: Weir polyinstantiates the internal and external app storage using a layered file system approach [Nei]. Our approach is similar to that used by Solaris [LS05] and most recently, Docker [Mer14]. Every secrecy context receives its own file system layer. That is, processes running in a particular secrecy context have the same view of the file system, which may be different from that of those running in other secrecy contexts. To maintain this separation, all file system operations are performed on the layer attached to a process, which then relays them to the underlying file system. Unlabeled processes are assigned the default (i.e., unlabeled) file system layer.

For efficiency, a layer only stores the changes made to the underlying file system (i.e., default layer) by processes running in the layer’s secrecy context. When a file present in
the default layer is first written by a process attached to a non-default layer, the file is first copied to the non-default layer and then modified. Future read and write accesses for the file are then directed to the copy. When a process attached to a non-default layer tries to read a file that has never been modified in the caller process’s layer, the read operation is performed on the original file on the default layer. *Weir*'s file system approach is an extension of its lazy polyinstantiation, i.e., new layers are created only when a process with a previously unknown secrecy context is initialized. Applications can transparently access storage using any file system API, and *Weir* directs the accesses to the correct files.

*Weir* stores the files copied to layers in layer-specific copy-on-write directories. While creating such directories, *Weir* accounts for the security and availability requirements of applications and users. For application-specific internal storage, the application-specific copy-on-write directories are created in an area that is accessible only to the particular application, and inaccessible to instances of other applications, even with equivalent labels. Whereas for public external storage, *Weir* creates common label-specific copy-on-write directories in an area accessible to all apps. This approach ensures that when an application is uninstalled, the data it wrote to external storage is still available to the user. We now demonstrate *Weir*'s approach with the following example.

**Example:** Figure 6.4 shows two instances of the component $C$, one of which is running in a process with an unlabeled secrecy context (i.e., label $\{\}$), while the other has a label $\{L_1\}$. *Weir* sets up a file system layer, i.e., $Layer (L_1)$, to mediate all file accesses by the labeled instance. Note that $Layer (L_1)$ is only attached to the processes with label $L_1$; processes with
other labels have their own separate layers. *Weir* does not allocate a layer for the unlabeled secrecy context.

As seen in Figure 6.4a, both the unlabeled and labeled instances of $C$ read from the shared preferences file (i.e., *SharedPrefs*). Further, *Layer* ($L_1$) relays all the read requests by the labeled instance of $C$ for unmodified *SharedPrefs* file to the default storage. Once the labeled instance of $C$ attempts to write to the *SharedPrefs* file, *Weir* copies it to *Layer* ($L_1$). For all future read or write accesses by instances with label ($L_1$), *Weir* uses the new copy of the *SharedPrefs* file stored in *Layer* ($L_1$). Note that the approach does not vary with the type of component accessing the file (e.g., content provider, service) or the kind of file being accessed (e.g., database, image).

**Security of copy-on-write directories:** *Weir* ensures the security of the layered directories through a combination of file-system labeling and Linux permissions. First, *Weir* prevents attempts by instances with lower labels to read files in the copy-on-write directories. This is done by initializing the secrecy label of such files to that of the layer when they are first written to the copy-on-write directory, and preventing processes with lower labels from reading them using the strict DIFC label check. Second, *Weir* prevents the implicit flows that may occur not due to the contents, but the presence, absence or the number (i.e., count) of such copy-on-write directories in private or external storage. To address implicit flows through the presence of specific copy-on-write directories, *Weir* uses random directory names that are only known to the system. To prevent implicit flows that make use of the number (i.e., count) of such copy-on-write directories, *Weir* creates the copy-on-write directories inside a parent directory owned by *Weir*. The Linux permissions of this parent directory are set to allow only write operations on its internal directories, and hence cannot be used to know the number of subdirectories.

Together with the polyinstantiation of application components and processes, *Weir’s* approach enables it to transparently polyinstantiate the file system, without modifying or affecting applications, satisfying Goal **G3**. For instance, *Weir’s* mechanism applies copy-on-write to a database file as it would to any other type of file, ensuring separation of data written by processes running in different secrecy contexts. Such transparency is absent in approaches such as Maxoid [XW15] that require modifications to content provider components and depend on the usage of specific database libraries.
6.5.3 Binder Enforcement and Label Changes

A component instance's label is implicitly set when it is instantiated. Similarly, a file's label is also initialized when it is first written to. For all successive accesses (i.e., future reads/writes to a file or direct Binder communication), *Weir* does not apply floating labels, instead applying strict DIFC label checks. Hence, any label changes after initialization can only be explicit, and not floating.

Applications aware of *Weir* may change the labels on their instances by raising them (to read secret data), or lowering them (to declassify data), provided the label changes are legal with respect to the data owner's policy for the security classes involved in the label change, as described in Section 2.3. We define the actual policy syntax in Section 6.6.1. For example, in order to read secret contacts labeled with label \{L_1\}, a component instance raises its label to \{L_1\}, and queries Android's *Contacts provider*. A new instance of the *Contacts provider* created with label \{L_1\}, with access to the storage Layer (L_1), retrieves the secret contacts wanted by the caller's instance. Similarly, an application may lower its label to declassify data for export to the network. We now describe the problem resulting from explicit label changes after initialization, and present our solution. Then, we discuss the security implications of our solution.

**Problem of label change after initialization:** Component instances may establish Binder connections with other component instances through the Activity Manager, and then use Binder RPC to directly communicate. When an instance changes its label after initialization, its established connections are be affected. That is, its new label may be higher or lower relative to the instances it is connected to.

A flow from a higher to a lower label violates DIFC secrecy constraints. Hence, *Weir* strictly enforces DIFC checks on Binder transactions to account for explicit label changes; i.e., *Weir's* kernel enforcement allows communication through Binder only when the receiver's label dominates (i.e., is higher than) the sender's. As a result, an explicit label change may make a component instance's current connections unavailable for communication. An explicit label change may also make the component instance's state inconsistent with its attached storage layer. At the same time, explicit label changes are unavoidable for applications that want to use *Weir*.

**Our approach:** To solve this problem, *Weir* provides applications with the *intent labeling*
mechanism, i.e., components can label calls (i.e., intent messages), before they are sent to the Activity Manager service, ensuring that the target component is instantiated with the label set on the intent. In fact, a component can polyinstantiate itself with the desired secrecy label by specifying itself as the target. Intent labeling eliminates the need for explicit label changes; the caller can start another component instance with the desired label without a change in its own label.

**Security implications:** Our design considers the security implications of intent labeling. **Weir** does not blindly trust the label set on the intent, as applications may otherwise abuse the mechanism for unauthorized declassification of data. For example, a malicious application in an instance with the label \{L_1\} may add confidential data to the intent, and set an empty label (i.e., \{\}) on the intent before calling itself with it. To account for such malicious use cases, **Weir** checks if the calling application would be authorized to explicitly change its current label to the label on the intent. **Weir** allows a call with a labeled intent to proceed only if the caller passes the check.

### 6.5.4 Domain Declassification

Problems with traditional network declassification are rooted in the decision to declare trust in the exporting subject, as discussed in Section 6.2. More precisely, in an internet-driven environment, it may be more practical for the data secrecy enforcement to reason about *where* the data is being delivered, rather than *who* is performing the export. **Weir** introduces the alternative of *domain declassification* to allow data owners to articulate trust in terms of the receiver, i.e., the target Web domain. **Weir** allows the data owner to associate a set of network domains \(t^D\) with its security class \(t\). When the data in context \(\{t\}\) is to be exported to the network, **Weir**'s enforcement implicitly declassifies \(t\), if the destination domain is in \(t^D\). The data owner is neither required to explicitly declassify nor trust the exporting application.

In Section 6.9, we discuss an example where the enterprise only wants data to be exported to a set of enterprise domains, irrespective of the application exporting it. Such a policy allows the user to use the same email application for both the personal and work account, but prevents accidental export of work data to the personal SMTP server. Domain declassification not only addresses the goal of practical declassification in a network driven environment (G4), but also prevents the user from accidentally exporting data from
a trusted application, but to an untrusted server.

Weir is not the first IFC system to use domains for declassification, although most prior systems to do so consider domains as security principals (e.g., COWL [Ste14], Bauer et al. [Bau15]). For instance, COWL confines JavaScript using a declassification policy analogous to the well-known same origin policy (SOP), i.e., code executes in the context of its origin, and hence possesses the declassification privilege for export to the origin’s Web domain. In this case, the origin Web domain is a first class security principal, as it has physical presence on the device in the form of the code running in its context. Thus, in COWL, the declassification privilege is still expressed in terms of the security principal that is sending the data (i.e., the origin). On Android, there is no direct correlation between Web domains and applications; i.e., Web domains do not have code executing in their context on the device, and hence are not security principals. Thus, Weir’s approach of expressing trust in the receiver of the data (i.e., the Web domain) rather than the sender is indeed unique among OS-level DIFC systems where Web domains may not be security principals [Alj12; NE13; Kro07; Zel06]. Hails [Gif12], an IFC web framework for user privacy, may be closer to Weir’s approach, as it allows users to declassify their data for specific domains. Hails users are prompted to explicitly declassify when network requests to disallowed domains are first made, which may not be feasible on Android (see Section 6.2).

Weir’s enforcement is limited to the device, and may not defeat an adversary controlling the network. While we leave this aspect relaxed for our threat model, we note that DNSSEC or IPsec could be used in such scenarios.

6.6 Implementation

We implemented Weir on Android Lollipop (version 5.0.1), and the Android Linux kernel (version 3.4). The source code can be found at https://wspr.csc.ncsu.edu/weir/. This section describes the essential aspects of our implementation.

6.6.1 Weir DIFC Policy

Weir derives its policy structure from the Flume DIFC model [Kro07], which consists of tags and labels. We define the basic policy structure in this section, while a detailed
description of the policy can be found in Appendix B.2.

A data owner (O) application defines a security tag $t$ (i.e., security class) for its sensitive data. A set of such tags forms a secrecy label ($S$). **Weir** enforces the classical IFC secrecy guarantee, i.e., “no read up, no write down” [BL73]. Information can flow from one label to another only if the latter dominates, i.e., is a superset of the former. For instance, data can flow from a process $P$ to a process $Q$ if and only if $S_P \subseteq S_Q$. Further, each tag $t$ has associated capabilities $t^+$ (for reading) and $t^-$ (for declassification). Data owners can delegate these to specific applications (i.e., to capability set $C_P$ for subject $P$), or all other applications (i.e., the global capability set $G$).

**Weir** extends Flume's syntax to support domain declassification through the capability $t^D$, which is a set of trusted Web domains specified by the data owner. To export data to the network ($S_N = \{\}$), a process $P$ must be able to change to an empty label (i.e., $S_P = \{\}$). That is, $P$ can connect to the network if $\forall t \in S_P, t^- \in C_P \cup G$. For all tags that may not be declassified in this manner, **Weir** implicitly declassifies the tags whose domain declassification capability contains the domain $d$ to which data is being exported.

### 6.6.2 Component Polyinstantiation and Compatibility

We provide a brief overview of **Weir**’s component polyinstantiation implementation. For the interested reader, a detailed implementation is available in Appendix B.1.

**Weir** modifies the Activity Manager service to implement its component polyinstantiation. When a component sends an intent message, the Activity Manager first resolves the identity of the target component to be called from the information in the intent and the static information present in the application manifest files. **Weir** does not interfere with this intent resolution process. Then, the Activity Manager chooses the actual runtime instance of the resolved component, which is where **Weir**’s polyinstantiation takes effect.

Since **Weir** does not affect the intent resolution described above, it is independent of the static configuration information in the Android manifests of applications. In fact, **Weir** even maintains the component-process mapping specified by the developer, as described in Section 6.5.1. **Weir** controls how components are instantiated, without modifying the components themselves. Hence, **Weir** can logically be said to be compatible with most developer options. Section 6.8.2 provides a compatibility evaluation for corner cases.
6.6.3 Initialization of Labeled Processes

On Android, the zygote process forks and prepares new processes for applications. When a new process is forked, Weir sets its secrecy abel in the kernel, and uses zygote to mount the appropriate storage layer to the process's mount namespace based on its label. We choose OverlayFS [Nei] for implementing the layers over other alternatives (e.g., aufs), as it is included in the Linux kernel (since version 3.18). As the current OverlayFS patch is incompatible with SELinux, we set SELinux to monitoring mode. This is a temporary limitation, as OverlayFS developers are working towards full integration [Wal15], which is on SELinux's Kernel ToDo list as well [Dev15]. Additionally, we could use a fine-grained block-level copy-on-write file system (e.g., BTRFS [Rod13]). There are advantages to using such file systems, as we describe in the trade-offs (Section 6.10). Note that while we could get the Android Linux kernel to compile with BTRFS, the build system support tools that are required to build Android's sparse-images for BTRFS (e.g., ext4_utils for ext4) are missing. Therefore, our prototype opts for OverlayFS, as it does not require user-space support.

If the process has a non-empty label, Weir separates the process's mount namespace from the global mount namespace using the unshare system call, and mounts the appropriate OverlayFS copy-on-write layer based on the label on top of the unlabeled file system. Processes initialized with empty labels are always allocated the unlabeled (lower) “default” layer. New layers are allocated when new labels are first encountered. Weir maintains a persistent mapping between a label, its assigned layer and the specific copy-on-write directories used for it on the internal and external storage.

6.6.4 Weir Kernel Enforcement

Weir uses a Linux security module (LSM) to create and maintain security contexts of processes and files in the kernel. The security context of a process contains its secrecy label, and positive and negative capabilities assigned at the time of creation, while that of a file only contains a secrecy label. We now describe the enforcement for file accesses, Binder communication and network access, as follows:
6.6.4.1 Files

*Weir* uses the *file_permission* LSM hook to mediate each file read and write access. The secrecy label of a file (stored in the xattrs) is initialized from the label of the process that first writes it. Strict DIFC label checks (and not floating labels) are performed for future reads/writes.

6.6.4.2 Binder

*Weir* mediates Binder transactions, including the transactions that transfer Binder objects and file descriptors, using the respective LSM hooks, and performs strict DIFC label checks involving the caller's and callee's labels. Further, *Weir* whitelists Binder communication to and from system services for compatibility. To prevent apps from using whitelisted services as implicit data channels, we manually analyzed all Android system services and their API, and modified services that may be used for implicit declassification. For example, we extend the Clipboard Manager service to provide label-specific clipboards. As we could polyinstantiate system content providers (e.g., Contacts Provider) without any modifications, we do not need to white list them.

6.6.4.3 Network

*Weir* mediates socket operations in the kernel, namely socket *start* and *bind*. All the tags in the calling process's label that cannot be declassified using its capability set are sent to *Weir*'s system service in the userspace for domain declassification via a synchronous upcall, along with the IP address of the destination server.

*Weir*'s system service then resolves the domain name from the IP address for domain declassification. This is challenging, as a reverse DNS lookup may not always resolve to the same domain used in the initial request. Fortunately, Android proxies all DNS lookups from applications to a separate system daemon. We modify that daemon to notify *Weir* when a process does a DNS lookup, including the domain name and the IP address that was returned. When the kernel does the upcall to the *Weir* service before allowing a connection to be established, this mapping is referenced to identify the destination domain. *Weir* allows the connection only if all the tags in the upcall can be declassified for that domain. This
mechanism is secure as an application attempting to do its own DNS lookup without using the proxy would be unable to add the domain’s entry to Weir’s mapping.

6.6.5 Tag Creation and Labeling API

Weir exposes an API for applications. Recall the motivating example, where OfficeEmail wants to protect an email attachment before sending it to a reader such as WPS Office. As shown in Figure 6.5, all OfficeEmail has to do is to create a tag that allows all applications to read, but only export to “officeemail.com”, and add it to the intent before sending the intent to start WPS Office. Note that while applications use absolute tag names (i.e., owner’s package name and the tag’s name), Weir generates a non-reusable random and globally unique 64 bit number for internal use. A separate internal representation provides efficiency, while allowing applications to assign variable-length tag names.

We envision a set of Weir management support applications in a production build of Weir. For instance, a Weir task manager to start/stop labeled component instances (i.e., manage state in memory), and a Weir data manager to provide the user with a centralized perspective over all labeled and unlabeled data (i.e., to manage state on storage). We leave the creation of such tools and their usability to future work.

6.7 Security of Polyinstantiation

Floating labels were first predicted to be prone to information leaks by Denning [Den76]. While language-level floating label IFC models (e.g., COWL [Ste14] and LIO [Ste11b; Ste15]) can mitigate such leaks, securely using floating labels is still a challenge for OS-level DIFC
Figure 6.6 Floating label DIFC system: Q receives 1 and guesses 0 for every reply not received.

systems (e.g., IX [MR92] and Asbestos [Van07]). We discuss an attack on an OS-level floating label DIFC system, described in Krohn and Tromer’s paper on the non-interference of Flume [KT09], and show how Weir is resistant to such data leaks. We use Android’s terminology to describe the attack.

We describe the attack twice; once in a floating label system without polyinstantiation (Figure 6.6) and once in Weir (Figure 6.7). Figure 6.6a shows the malicious components (e.g., services) $P$, $Q$, $Q_1$, and $Q_2$. $P$ has obtained the data “01”, and the accompanying label $\{L_1\}$. $P$ wants to transfer the data to $Q$, without $Q$ obtaining the label $\{L_1\}$. Note that $Q$, $Q_1$, and $Q_2$ initially have the empty label $\{}$. Additionally, $P$ and $Q$ have a prior understanding that $P$ will call the $i$th service of $Q$ to indicate “0” at the $i$th bit. $Q$’s components are programmed to send a message to $Q$ after a predetermined time if they do not receive a message from $P$ (i.e., indicating a “1”). Since the first data bit is “0”, $P$ sends “0” to $Q_1$, whose label floats to $\{L_1\}$ (Figure 6.6b). After a predefined time, the component that did not receive a message from $P$, i.e., $Q_2$, sends a “1” to $Q$ (Figure 6.6c). The data leak is successful, as $Q$ knows that the second bit is “1”, and assumes the first to be “0”, all without acquiring the label $\{L_1\}$. As Android does not place any limits on the number of components, a wider $n$ bit channel is possible with $n$ components.

Weir’s polyinstantiation defeats this attack by creating a new instance of $Q_1$ in a separate process to deliver a call from a label that mismatches its own, as shown in Figure 6.7a. Next, the unlabeled instance of $Q_1$ and $Q_2$ both call $Q$ with data “1”, as shown in Figure 6.7a. In fact, for $n$ components of $Q$, $Q$ will always get $n$ calls with data “1”, as Weir will polyinstantiate all the components that have been called by $P$ with the label $\{L_1\}$. Weir’s use of floating labels is resistant to implicit flows inherent to regular floating labels, as labels do not float.
to the original instance, but to a new instance created solely for the call with the different label. Note that attacks on floating labels are not applicable to direct calls through Binder, where strict DIFC label checks are applied instead of floating labels.

Jia et al [Jia13] attempt to solve a similar problem in their floating label based model, by making the raised label the component’s base (i.e., static) label. This ensures that future instances of the component are bound to the static label, rendering it inaccessible in all other contexts. This solution is specific to the attack they discuss, and does not work in other cases. For instance, in our example from Figure 6.6c, raising the static label of $Q_1$ static still allows the current data leak, but only prevents the use of $Q_1$ for future leaks, reducing the channel width. Attackers who are cognizant of this fact can coordinate the components to be used for every attack, and will be able to transfer a significant amount of data before all the components have restrictive static labels.

### 6.8 Evaluation

We performed experiments to evaluate the performance and compatibility of Weir. The experiments were designed to answer the following questions:

**Q1** Is *Weir* compatible with developer preferences that manipulate component instantiation?

---

3For brevity, we do not describe the attack here, but refer the reader to page 8 of the paper by Jia et al. [Jia13]
Q2 What is Weir’s performance overhead?

Q3 Is Weir scalable for starting a component?

We now provide an overview of the experiments and highlight the results. The rest of this section describes each experiment in detail, including the methodology and observations.

6.8.1 Experiment Overview and Highlights

Weir’s polyinstantiation does not modify the applications/components themselves or their configurations, but only modifies the manner in which they are instantiated. Therefore, we need to evaluate the compatibility of the configuration options that directly manipulate a component’s instantiation, leading to question Q1. To answer this question, we tested compatibility with configurations that allow developers to control a component’s instantiation, namely the singleTop, singleTask and singleInstance launch modes described in Section 2.2. We triggered the launch modes in popular Android applications from the Google Play store, and observed the behavior of labeled as well as unlabeled instances of each application. During our tests we did not observe any application crashes or unexpected behavior. Every launch mode worked as expected, while the underlying polyinstantiation ensured delivery of calls to instances with the caller’s label.

We measured the performance overhead of Weir over an unmodified AOSP build of Android to address question Q2. Our tests involved common operations, such as starting components, processes, reading/writing files, and connecting to the Internet. We compared results between an unmodified AOSP build, Weir with an unlabeled application, and Weir with a labeled application. Our results in Table 6.1 show negligible overhead for both the unlabeled and labeled instance for most operations. Even in cases where the overhead percentage is large for starting some components (e.g., services), the absolute overhead value is negligible (less than 4ms).

Android’s mechanism of resolving a call to a component first retrieves that target’s identity using the static component “names” from the information in the application manifest files. Then it delivers the call to one of the running instances of the component. We can say that a component’s start time is affected by 1) the number of total static components on the device (unaffected by Weir), and 2) the number of its own running instances (manipu-
lated by Weir’s polyinstantiation). The insight here is that multiple running instances of a component (due to Weir) only affect its own start time, and not that of other components. Therefore, we evaluate scalability of starting components with polyinstantiation in place on a per-component basis. To address question Q3, we measured the performance overhead incurred in starting a single component, when a large number of its instances already exist in the system. Our results in Figure 6.8 show a linear increase in the start time with increase in the number of simultaneous component instances. The absolute overhead value for starting a component with 100 simultaneous instances is about 42ms.

6.8.2 Compatibility with Activity Launch Modes

6.8.2.1 Dataset

For each launch mode, we randomly select 10 applications from the top 50 applications (all categories) from the Google Play store. We have included the list of applications tested for compatibility in Appendix B.3.

6.8.2.2 Methodology

We test each launch mode as follows. We first launch each application from two separate unlabeled components, and navigate to the specific activity we want to test. With this step, we confirm that the application and specifically the singleTask/Top/Instance activity works as expected. We then start the same application from a labeled context, without closing the existing instances, and navigate to the same activity. We record any application crashes or unexpected behavior (e.g., a singleTask activity being launched in a separate task if called from a labeled context).

6.8.2.3 Observations

We did not observe any unexpected behavior. Activities started in their assigned tasks. In the case of singleTask and singleInstance activities, two runtime instances of the same activity ran in the designated task instead of one; i.e., one labeled and the another unlabeled. Intents were delivered to the activity instance whose label matched the caller’s. The activity at the top in the recent task, or the activity presented to the user, was the one that was most
Table 6.1 Performance - Unmodified Android (AOSP), Weir in unlabeled context, Weir in labeled context.

<table>
<thead>
<tr>
<th>Operation</th>
<th>AOSP (ms)</th>
<th>Weir w/o label</th>
<th>Weir w/ label</th>
<th>Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity start</td>
<td>20.06±4.47</td>
<td>22.22±4.69</td>
<td>20.82±4.87</td>
<td>2.16 (10.77%)</td>
</tr>
<tr>
<td>Service start</td>
<td>13.94±2.87</td>
<td>14.96±2.85</td>
<td>17.36±4.78</td>
<td>1.02 (7.32%)</td>
</tr>
<tr>
<td>Broadcast Receiver start</td>
<td>12.92±3.96</td>
<td>11.42±4.44</td>
<td>11.86±3.34</td>
<td>-1.5 (-11.6%)</td>
</tr>
<tr>
<td>Content Provider start</td>
<td>4.54±2.28</td>
<td>7.26±5.32</td>
<td>7.9±4.73</td>
<td>2.72 (59.91%)</td>
</tr>
<tr>
<td>Process start</td>
<td>127.18±5.62</td>
<td>130.28±5.63</td>
<td>132.98±6.66</td>
<td>3.1 (2.44%)</td>
</tr>
<tr>
<td>File Read (1MB)</td>
<td>42.38±6.05</td>
<td>43.46±5.44</td>
<td>41.32±5.39</td>
<td>1.08 (2.55%)</td>
</tr>
<tr>
<td>File Write (1MB)</td>
<td>46.8±5.79</td>
<td>47.84±5.42</td>
<td>47.16±5.85</td>
<td>1.04 (2.22%)</td>
</tr>
<tr>
<td>Internet</td>
<td>66.98±3.62</td>
<td>65.68±2.78</td>
<td>69.00±7.04</td>
<td>-1.3 (-1.94%)</td>
</tr>
</tbody>
</table>

recently delivered an intent resulting from the user’s action. This behavior is compatible with singleTask and singleInstance activities, and also maintains label-based separation in memory. The results were consistent between all applications in the dataset.

6.8.3 Microbenchmarks

We evaluate *Weir’s* performance on a Nexus 5 device, measuring the operations most affected by *Weir’s* polyinstantiation and domain declassification, i.e., component and process initialization, file access, and network access. We perform 50 instances of each experiment, with 200 ms between subsequent runs. Table 6.1 shows the mean with 95% confidence intervals. We now discuss the specific methodology and observations for each operation evaluated. Cases with negative overhead can be attributed to the high error in some operations.

6.8.3.1 Component and Process start

We measure the start times for components, i.e., the time from the placement of the call (e.g., startActivity) until it is delivered to the component. The component is stopped between measurements. To measure the process start time, we kill the process between subsequent measurements. While the overhead percentages may be high for some components (e.g., providers), the absolute values are low, and would not be noticeable by a user. Further, the process start time shows minimal overhead, which demonstrates the efficiency of the file system polyinstantiation in zygote.
6.8.3.2 File access

We perform file read and write operations on a 1MB file using a 8KB buffer. Since the entire check is performed using the process and file labels in the kernel, the overhead value is negligible (e.g., about 0.77% for a labeled file write). Further, we measure the cost of copying the 1MB file to the labeled layer, i.e., when the file is first written by the labeled instance, by repeating the file write experiment on Weir but deleting the file between runs. The extra time taken to copy relative to AOSP is 5.98 ms, or less than 13% overhead. While this cost may increase with file size, OverlayFS is scalable even for large files (e.g., Docker containers), as demonstrated by RedHat’s the evaluation of OverayFS for Docker [Jer].

6.8.3.3 Network access

We measure the time taken to establish a network connection using the commonly used HTTPSURLConnection API. The application initiating the connection is labeled with a tag which does not allow explicit declassification, but uses the domain declassification capability. The overhead (2.02 ms or 3.02%) includes the time taken by the DNS proxy to inform Weir of the lookup, as well as two synchronous kernel upcalls (socket connect and bind). For the unlabeled instance, there is no overhead as there are no upcalls and the DNS proxy communication has negligible overhead.

6.8.4 Scalability of Component Polyinstantiation

6.8.4.1 Methodology

We create up to 100 simultaneous instances of a service component, each with a different label. We then invoke the last instance, i.e., from a caller with the last instance’s label, and measure the component start time. We perform the experiment for 10 through 100 simultaneous instances, incrementing the number of instances by 10 at each step. Note that this experiment presents the worst case scenario; i.e., our prototype does not implement any particular strategy (e.g., least recently used (LRU)) for matching components from a list of available instances, and a request with the last component instance’s label will always result in the label being compared with all available instances.
Figure 6.8 Time to start a component when 0 → 100 instances (in steps of 10) already exist.

### 6.8.4.2 Observations

Figure 6.8 shows a linear increase in service start time with increasing component instances. Further, using the time required to start an unlabeled service instance from Table 6.1, we observe that starting an instance out of 100 simultaneous labeled instances incurs an additional overhead of approximately 42 ms. While the percentage overhead might be high, the low absolute value for even the 100 instances case demonstrates the scalability of our approach.

### 6.9 Case Study

We investigated the use of labeled enterprise data with an unmodified third party email (K-9 Mail) application [dev15]. With this case study, we demonstrate Weir’s utility, and motivate the trade-off discussion in Section 6.10.

**Application Setup:** We created an enterprise cloud application, BCloud that allows the user to sync her work data (e.g., contacts, documents) to the device. Further, we used the popular email application K-9 Mail with both user and enterprise data. The setup is as follows:

1. **BCloud.** We assume that the enterprise policy is to enable the use of third party applica-
// Creating the tag ‘t’
domains={"www.bcloud.com", "smtp.bcloud.com", ...};
createTag(‘t’, domains);

Figure 6.9 BCloud’s policy configuration

addTag(‘t’); // raise own label to {t}
// perform sharing action ...
removeTag(‘t’); // lower own label

Figure 6.10 BCloud raises its label

ations with work data, but to allow export to only enterprise domains. For example, work data must only be emailed using the work SMTP server smtp.bcloud.com. Thus, BCloud creates a tag t as shown in Figure 6.9. To set the policy before sharing its data or saving it to storage, BCloud may temporarily raise its label to {t} (Figure 6.10), or start itself or other applications with {t} using intent labeling (Figure 6.11). For instance, BCloud raises its label before copying the work contacts to Android’s Contacts Provider.

2. K-9 Mail. We configured K-9 Mail for both personal and work email accounts. Like most modern email clients, K-9 Mail allows the user to send an email using the work or the personal account, using the send as email field. Internally, K-9 Mail uses the SMTP server smtp.gmail.com for the personal account, and smtp.bcloud.com for the work account. To assist the user in composing an email, K-9 Mail retrieves contacts from the Contacts Provider app, and makes suggestions as the user types into the “to” (i.e., sender) field.

Experiment: We opened a document from BCloud in the WPS Office application. Then, from the WPS Office app, we shared the document with K-9 Mail. K-9 Mail’s “compose” window was displayed. We then chose to send as the work account, and picked a contact to add to the “to” field. We tried to attach another file, and the “attach” action opened Android’s system file browser. We selected a file and returned to K-9 Mail’s compose screen. We then switched to the home screen without sending the work email. We repeated the entire experiment in the default (i.e., unlabeled) context, with the send as field set to the personal account. We then sent both emails. Throughout the experiment, we watched the

4We used mail.yahoo.com, smtp.mail.yahoo.com and imap.mail.yahoo.com as BCloud’s trusted domains.
system log for important events (e.g., network denial).

**Observations:** We made the following observations, and verified them using the system log:

1. **Context-specific instances.** As we shared work data (in the context \{t\}) with *WPS Office* and subsequently *K-9 Email*, instances of these applications (i.e., processes and components) were started in the work context \{t\}, and attached to the internal and external (SD card) storage layer *Layer\(_t\)*. The unlabeled context resulted in separate instances with the empty label (\{\}), attached to the default storage layer. Instances in both contexts existed concurrently, without any crashes or abnormal behavior.

2. **Context-specific data separation.** While attaching another document in the work (\{t\}) instance of *K-9 Mail*, we could see all the documents on the default storage layer (i.e., unlabeled files), and documents in work *Layer\(_t\)* (i.e., added from *BCloud*). On the contrary, in the default context, we could only see the files on the default layer. Further, in the default context, *K-9 Mail* suggested from all of the user’s unlabeled contacts, but none of the work contacts. In the work context, *K-9 Mail* suggested from all the work contacts, and the unlabeled contacts that existed before *BCloud* synced its labeled contacts. That is, *K-9 Mail* could not see new records created in the default layer’s contacts database after it was copied over to *Layer\(_t\)*.

3. **Domain Declassification.** In the work context, *K-9 Email* was unable to connect to the SMTP and IMAP sub-domains of *gmail.com*, but could only connect with the domains declassified by tag \(t\). Unmodified *K-9 Email* silently handled these network access exceptions, without crashing or displaying errors messages.
6.10 Trade-offs and Limitations

This section describes the trade-offs of our approach, motivated in part by the observations in the case study.

1. **Centralized perspective**: The user cannot view both labeled and unlabeled data together, unless an application is started in the labeled context (e.g., *K-9 Mail* in context \{t\}). We envision modified application launchers and phone settings that allow the user to start applications (e.g., File Browsers) with a certain label by default, for making labeled and default data available together. Our test apps use similar techniques; hence such launchers should not be hard to create. On the other hand, a centralized perspective on more than one non-default context (e.g., \{t_1, t_2, t_3, ..\}) may require a trusted OS application exempt from polyinstantiation (but subject to only floating labels), as floating labels by themselves are vulnerable to information leaks (Section 6.7).

2. **Updates to default layer**: While context-specific versions of files may be generally acceptable, in case of database files (e.g., contacts read by K-9 Mail in the work context) the user may expect new records in the unlabeled context to be propagated to the copy in the labeled context. The lack of updates is mainly a trade-off of our file-level copy-on-write implementation (i.e., OverlayFS). As mentioned in Section 6.6, a block-level copy-on-write file system (e.g., BTRFS [Rod13]) may mitigate this trade-off, as it would only copy the blocks modified by the labeled context, and newly allocated blocks in the default context would be accessible to the labeled context, although this aspect needs further exploration.

3. **Access control denials**: Floating labels ensure that inter-component communication is never denied, and that resources (e.g., files, other components) are available in all secrecy contexts. Although apps may be denied network access, research has addressed this challenge in the past (e.g., AppFence [Hor11]). Further, most IDEs (e.g., Eclipse) enforce compile-time checks for proper exception handling, and it is uncommon for apps to crash due to network denial, as observed in the case study as well.

4. **Instance Explosion**: *Weir* creates separate context-specific K-9 Mail instances, *only for the contexts in use*. The theoretical worst-case count of component instances is equivalent to the number of components multiplied by the number of all existing contexts [DP10]. Our event-based and “lazy” instantiation makes this worst case practically improbable, unlike
approaches that execute *all existing* contexts. On the other hand, a denial of service attack on a particular application component may be feasible, by starting a very large number of its instances in a short amount of time for noticeable impact on the lookup time of that component. Our implementation can be modified to detect and prevent unusual rates of component instantiation. Note that polyinstantiation of a component only affects its own lookup time, and cannot be used for an attack with a device-wide impact.

5. **Resource Overhead:** Polyinstantiation may cause resource overhead in terms of the memory, battery and storage. The memory overhead is manageable as Android’s out of memory manager automatically reclaims memory from low priority components. Further, any measurement of the battery or storage use is bound to be subjective with respect to the number of labels, number of apps/components, type of apps (e.g., game vs. text editor), aspects of the user scenario (e.g., user-initiated flows, scenario-specific storage access). An objective large-scale study will be explored in the future.

6. **Consistency Issues:** To a remote server, the instances of an application in Weir are analogous to instances running on different devices (e.g., a user logged in from two devices). Hence, any data consistency issues in such scenarios are not a result of polyinstantiation.

7. **Covert Channels:** Weir mediates overt communication between subjects and objects, but does not address covert channels existing in Android. A clearance label [BL73; Van07; Ste11b] can be used to defend against adversaries using covert channels by preventing access to certain tainted data in the first place. While a clearance label can be easily incorporated into Weir, setting the clearance policy for third party applications with unpredictable use cases is hard, and needs further exploration from a policy specification standpoint. Finally, unlike IFC systems that focus on preventing untrusted code within a program from exfiltrating data (e.g., Secure multi-execution [DP10]), Weir’s focus is inter-application data sharing. Hence, compartmentalizing an application using clearance is outside the scope of this work.

8. **Explicit labeling of messages and files:** On Android, an indirect message through the OS (e.g., intent message) is required before a bi-directional Binder connection can be established between two instances. Weir allows floating labels on such indirect communication (but not on direct Binder calls), and polyinstantiation ensures that the two instances at the end of a bidirectional Binder connection have the same label, which is sufficient for
synchronous Binder messages. Hence, labeling of individual Binder messages does not provide additional flexibility, unlike in explicit labeling DIFC systems (e.g., COWL [Ste14], Flume [Kro07]). Note that Weir allows explicit labeling of indirect messages (i.e., intent labeling). Further, explicit labeling of a file with a label that is different from its creating process instance would place it on an incorrect layer. Such incorrectly stored files will not be visible to future instances started with matching labels, and may cause unpredictable application behavior. Thus, our design trades the flexibility in explicitly labeling files for stable context-sensitive storage.

6.11 Summary

In this chapter, we describe how Android's component and storage abstractions make secure as well as practical DIFC enforcement challenging. We present the mechanism of “lazy” polyinstantiation, which makes label propagation in DIFC sensitive to the secrecy context, in addition to making all resources available in every context. Additionally, we present domain declassification to allow data owners to express trust in remote domains. We design and implement Weir, a practical DIFC system for Android. Weir shows a negligible impact on performance, and maintains compatibility with legacy applications. Hence, with Weir, we demonstrate secure as well as practical DIFC enforcement on Android.
In the era of pervasive computing, security of user data and resources is of paramount importance. Increasingly complex systems such as IoT platforms (e.g., IFTTT [TT10] and Samsung SmartThings [Sam]), smartphone platforms (e.g., Android and iOS), and even traditional commodity platforms are being leveraged for processing and managing user data. However, our knowledge of policy specification has not kept pace with the rise of complex systems that are increasingly relying on the user to specify the security policy.

Further, user data has become increasingly user-specific. Users no longer directly deal with generic files, but create specific data objects such as notes, whiteboard snapshots, and selfies. This data is abstract, i.e., its importance and properties are subjective. System
designers and application developers cannot specify a security policy for abstract user data. The situation is even critical for novel security systems that provide strong data security guarantees for user-specific data (e.g., decentralized information flow control (DIFC) systems for Android [NE13; Jia13; XW15; Nad16], Chromium [Bau15]). Such systems are impractical to deploy unless users specify security policies; and users are bad at specifying security policies [Sad09; MR05] without assistance.

This chapter raises the simple but important question of policy specification: how to teach the system what the user wants to protect, and how the user wants to protect it? Consider the following example: a smartphone user wants to synchronize all personal notes with her cloud account, except notes labeled as medical data. Since we are dealing with user-specific data-use scenarios, we can justifiably expect the user to provide some input to the system. However, expecting the user to enumerate every possible scenario involving medical data is impractical. The policy must be predicted.

We propose the approach of specifying Policy by Example (PyBE) for user-specific data. PyBE is inspired by the successful use of programming by example (PBE) for program synthesis. We emulate the approach of Gulwani [Gul11], where the user specifies examples consisting of the input and output, and the system learns a program that can predict the output for unknown (but similar) inputs. Similarly, in PyBE, the user specifies policy examples, in terms of the data-use scenario (i.e., the input) and the policy decision (i.e., the output). The system uses the user-specified examples to predict policy decisions for new scenarios. By requiring only relevant examples, and not complete policy specification, PyBE makes policy specification tractable.

Predicting security policies for abstract, user-specific data with unknown properties is hard. Since the data is abstract, our learner cannot make any assumptions about the input data points that may assist in prediction. In contrast, prior work on predicting privacy policies for well-known private data [Kel08; Cra11] can make assumptions that aid prediction; e.g., Cranshaw et al. [Cra11] take advantage of probabilistic models to learn location privacy policies knowing that location and time are continuous variables. PyBE cannot make any such assumptions, which puts us at a significant disadvantage. However, this disadvantage drove us to embrace a simpler approach that does not demand specific properties from data.

We chose a variant of the k nearest neighbor (kNN) classifier [Mur12] for predicting
policies. Our key requirements were that the algorithm must be (1) non-parametric, i.e.,
independent of probabilistic models that rely on fixed set of parameters, and (2) easy to
explain, i.e., for the user to understand how the policy was inferred. Recall that a policy
example is composed of a scenario and the policy decision for that scenario. For predicting
the policy decision for a new scenario, our algorithm performs a nearest neighbor search for
finding similar scenarios from the user's policy examples, and predicts the policy decision
of the majority.

An important challenge in applying kNN is calculating the distance between data points.
For calculating the distance between policy examples, we treat policy examples (i.e., the
scenarios) as Boolean functions, and propose a novel distance metric for Boolean functions.
Further, we recognize that some policies may be more important to the user than others.
Therefore, we extend our metric to support weights. Note that existing distance metrics
(e.g., jaccard distance) would require significant re-engineering to incorporate weights,
which is a motivating factor behind developing a new metric.

PyBE recognizes that policy specification by users in any form is error prone. A key
contribution of PyBE is our use of active learning for enabling the user to correct policy
decisions. We draw inspiration from the work of Gulwani [Gul11], which detects noise in the
user's examples, and uses an interactive approach to prompt the user for new outputs for
problematic examples. Similarly, we use noise in the user's policy examples as an indication
of potential errors in policy decisions, and suggests changes to the user. We define noise
for policy examples in terms of invariants on the nearest neighbor graph.

We evaluate the feasibility of our approach with a study of expert users. Our study
involves 8 participants, and 5 target security policies. As a result, we solve 40 independent
policy specification problems. Our participants generate 246 policy scenarios in total (30
on average), and assign policy decisions for 5 policies, resulting in a total of 1,230 policy
examples across users. We perform two experiments with this data. First, we compare the
errors in policy decisions found using a manual review and a PyBE-assisted interactive
review of policy examples. Then, we test PyBE’s prediction for randomly generated scenarios
with unknown policy decisions.

PyBE demonstrates a prediction accuracy of over 76% across all users. More importantly,
PyBE fares better on average than our assumed baseline of a random coin flip, and a naive
approach of predicting the majority decision from the training data. A significant finding is
that our interactive review approach (i.e., using active learning) helps participants find *five times* as many errors as their manual reviews.

We summarize the contributions of this chapter as follows:

- We introduce the Policy by Example (PyBE) paradigm for predicting user-specific security policies. Our approach takes labeled policy examples from the user, and predicts labels (i.e., decisions) for new policy scenarios, thereby making policy specification a tractable problem. We propose a novel distance metric to apply kNN to policy examples, and extend it to support weights.

- We use an interactive approach to assist users in finding incorrect policy decisions in their examples. We empirically demonstrate the effectiveness of our approach over manual policy reviews.

- We perform a feasibility study with expert users, and demonstrate 76% prediction accuracy on average. We demonstrate that our approach performs better than both a baseline as well as a naive approach.

This chapter is the first step in our vision of a policy assistant for user data. With PyBE, we provide an approach for predicting security policies for user-specific data, and demonstrate its technical feasibility. Further, we analyze our incorrect predictions, and describe the lessons we learned in the process. Finally, we describe challenges (e.g., usability for non-experts, modeling policy change) and future research directions in this promising new area.

### 7.1 Motivation and Problem

User data and data-use scenarios are user-specific. External observers such as system designers or application developers cannot specify the user's security policy without knowing the user's context of data use [Nis04; Bar06]. Moreover, this constraint is not limited to user-owned data; prior work demonstrates that even the security preferences for enterprise data vary with users and personal data-use contexts [GS10].
Our motivating example demonstrates how two users may differ in terms of the relevance of data-use scenarios as well as security preferences for the same scenarios. We then describe the problem addressed by this chapter.

7.1.1 Motivating Example

Alice and Bob are two smartphone users, who use a fictional note-taking application *Notes* (similar to Google Keep) on their smartphones to collect and organize information. Consider two data-use scenarios for Alice and Bob, as follows:

**Scenario 1**

Alice consolidates expenses by scanning paper receipts into the *Notes* application, which then syncs the receipts with Alice’s cloud provider (such as Google Drive). Alice trusts the cloud with most information, with the exception of medical data. Hence, when Alice scans a receipt from the hospital, she still wants to store it locally with *Notes*, but does not want it exported to the cloud.

Similarly, Bob uses *Notes* to aggregate all documents, i.e., data created or scanned using the smartphone and saved as documents using *Notes*. Bob’s notes are backed up to Bob’s enterprise cloud by default. Bob wants to backup all notes, except personal documents created after work hours.

**Scenario 2**

Alice and Bob meet at a conference and exchange business cards. As Alice is self-employed, she feels confident in backing up all business cards acquired after work hours to her enterprise cloud. Bob, on the other hand, does not want to disclose networking opportunities to his company. Hence, Bob does not want business cards collected after work hours to be backed up to the enterprise cloud.

**Observations:** We make two observations from the scenarios:

O1 *User data and data-use scenarios are diverse.* As seen in Scenario 1, different users may have different answers for what scenarios/conditions are applicable to them.
Alice cares about medical receipts, which may not be relevant to Bob, who in turn cares about personal documents.

**O2** Security preferences for user data stem from the user’s personal circumstances. Even if two users face the same scenario, they may not make the same decision on how they want to protect their data. In Scenario 2, Alice and Bob both collect business cards after work hours, but make different choices based on their respective employment situations.

### 7.1.2 The Problem

In this chapter, we focus on the problem of specifying user-specific security policies. The nature of the problem dictates that the policy specification must receive input from the user. However, it is impractical to expect the user to specify the policy for every scenario in an exponential space. Hence, this chapter addresses the problem of predicting the security policy for new data-use scenarios, based on the scenarios previously described by the user.

### 7.2 Policy by Example (PyBE)

PyBE is inspired by the programming by example (PBE) approach for program synthesis [Gul11], which learns a program from input-output examples. As shown in Figure 7.1a, the user provides policy examples (i.e., in terms of data-use scenarios and policy decisions), and PyBE interactively suggests corrections to the user's policy decisions. As shown in Figure 7.1b, PyBE then predicts policy decisions for new scenarios.

This section provides the overview and the intuition behind our approach. We describe
Table 7.1 Bob's example policies for the WorkCloud policy.

<table>
<thead>
<tr>
<th>No.</th>
<th>Scenario</th>
<th>Policy Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{Home, Photo}</td>
<td>deny</td>
</tr>
<tr>
<td>2</td>
<td>{Work, Photo}</td>
<td>allow</td>
</tr>
<tr>
<td>3</td>
<td>{Document}</td>
<td>allow</td>
</tr>
</tbody>
</table>

PyBE more formally in Section 7.3. We start by describing the structure of a policy example.

### 7.2.1 The Policy Example

A policy example is composed of a scenario, and a policy decision (i.e., allow/1 or deny/0) for that scenario. A scenario is as a set of tags, where each tag denotes the resource to be protected (e.g., business card) or a condition that influences the policy (e.g., created after work hours). Using a set of tags enables users to describe complex scenarios composed multiple conditions or data objects. Our use of tags is motivated by prior work that demonstrates that users can effectively re-purpose organizational tags to express access control policies [Kle12].

In addition to the user-customizable policy example, we also define a fixed policy target which represents the action controlled by the policy; e.g., exporting data to the enterprise cloud, which we name the WorkCloud policy target. Policy specification is performed separately for each policy target, i.e., independent of other targets. Thus, each target represents a separate high-level policy that must be specified (e.g., the user's WorkCloud policy). The policy targets used in this chapter are motivated by prior work on restricting the export of secret data to the network [Ste14; Bau15; Nad16].

Table 7.1 shows Bob's policy examples for the WorkCloud policy target. We describe each example, along with Bob's security requirement associated with it. First, Bob considers data created at home to be personal, so Bob's photos created at home must never be exported to the enterprise cloud. Thus, Bob denies export for example 1, i.e, {Home, Photos}. Second, photos taken at work may be exported to the enterprise cloud. Hence, Bob allows export for example 2, i.e., {Work, Photos}. Third, Bob does not (currently) imagine a situation where he would deny export for documents. Hence, Bob allows export for example 3, i.e., {Document}. We now use Bob's examples to informally describe PyBE. Our algorithm is
formally described in Section 7.3.

7.2.2 Our Approach

As described previously, PyBE uses a variation of the kNN algorithm for predicting policies. That is, Bob provides PyBE with a set of policy examples (i.e., scenarios as well as decisions). When faced with a new example, i.e., a new scenario with an unknown policy decision, we perform a nearest neighbor search of Bob's examples. That is, we search Bob's examples for the examples closest to the new example, and predict the policy decision of the majority of the closest examples. Note that distance between examples is described in terms of their scenarios (i.e., when we say “examples are close”, it means their scenarios are close).

An approach for predicting security policies should be deterministic if we want users to understand its outcome (i.e., independent of arbitrary parameters). Based on this rationale, we eliminate the need to specify the parameter $k$. Our variation of kNN considers the closest neighbors as all neighbors at the closest distance, instead of $k$ neighbors at varying distances.

We now demonstrate our approach with a manual walk-through. A manual walk-through is feasible because the basic process of kNN is intuitive and its outcome is easy to explain. To demonstrate our approach, we predict policy decisions for the following new scenarios for Bob: \{Home\} and \{Home, Document\}, using Bob's initial policy specification shown in Table 7.1.

Consider the first new scenario, \{Home\}. Just by looking at Bob's specification in Table 7.1, the reader may identify example 1 (i.e., \{Home, Photo\}) as closest to the new scenario, since it is the only example that includes the tag \texttt{Home}. As a result, we predict the policy decision for the new scenario \{Home\} as deny, i.e., as the decision of its nearest neighbor \{Home, Photo\}. This decision mirrors Bob's assumption of data created at home being personal, and not exportable to the enterprise cloud. PyBE's distance metric described in Section 7.3 uses a similar property for computing distance between two examples, and comes to the same conclusion.

Now consider the second new scenario, \{Home, Document\}. This time, there are two examples that seem to be equally close to the new scenario, i.e., \{Home, Photo\} and \{Document\}, since they each have one tag in common with \{Home, Document\}. Since both the nearest
Table 7.2 Bob's extended set of examples for the WorkCloud policy target, with newly added examples in **bold**.

<table>
<thead>
<tr>
<th>No.</th>
<th>Scenario</th>
<th>Policy Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{Home, Photo}</td>
<td>deny</td>
</tr>
<tr>
<td>2</td>
<td>{Work, Photo}</td>
<td>allow</td>
</tr>
<tr>
<td>3</td>
<td>{Document}</td>
<td>allow</td>
</tr>
<tr>
<td>4</td>
<td><strong>{Home, Document}</strong></td>
<td>deny</td>
</tr>
<tr>
<td>5</td>
<td>{Home, Memo}</td>
<td>allow</td>
</tr>
</tbody>
</table>

examples have different policy decisions, our simple metric is insufficient. This is one of the motivations for introducing weights. Suppose Bob considers personal data created at home (i.e., the tag Home) to be most confidential. Therefore, Bob assigns Home more “importance” (i.e., a higher weight) than any other tag in terms of its influence on the policy decision. As a result, the new scenario {Home, Document} can be deemed closer to {Home, Photo} than {Document}, as Home has a higher weight and more say in the decision than the other tags, e.g., Document. Thus, export is denied for {Home, Document}, which aligns with Bob’s preference of data created at home being personal, and not exportable to the enterprise cloud.

The purpose of weights is not limited to breaking ties. Suppose Bob specifies another example {Document, Receipt} with decision allow. Now consider another new scenario {Document, Receipt, Home}. Without any knowledge of weights, it is easy to see that {Document, Receipt} would be the example closest to the new scenario {Document, Receipt, Home}, resulting in allow being predicted (i.e., there is no tie). At the same time, we know that Bob has allocated a higher weight to Home, since Bob considers home data to be confidential and important with respect to the WorkCloud target. The weights ensure that {Document, Receipt, Home} is closer to {Home, Photo} instead of {Document, Receipt}, and export is denied as per Bob’s actual security preference. Simply stated, weights enable the user to make some information tags beat others in the distance calculation. Our weighted metric described in Section 7.3.2 uses a similar reasoning.

An important contribution of PyBE is that it recognizes that policy specification by users can be error-prone. PyBE uses active learning to engage the user in finding and correcting potential errors in their policy decisions. Our approach is inspired by the work of
Gulwani [Gul11], which detects noise in the user’s input-output examples, and recommends the user to change the incorrect output. Similarly, PyBE looks for noise in the user’s examples, which may indicate one or more potentially incorrect policy decisions. We use our variant of kNN for this purpose. Note that the objective of this task is to engage the user in finding errors in existing examples, and not to predict policy decisions for new examples. We explain our approach with the following extension to Bob’s policy:

Suppose Bob adds two additional examples, i.e., \{Home, Document\} with decision deny, and \{Home, Memo\} with decision allow. We borrow the first example (i.e., for a document created at home) from the previous discussion on weights. The second example shows Bob’s policy for a memo created at home. Further, recall that Home has a higher weight, hence examples containing Home will be closer to each other than other examples not containing Home. Bob’s complete set of examples is shown in Table 7.2.

We perform a nearest neighbor search for the example \{Home, Memo\}, and identify \{Home, Photo\} and \{Home, Document\} as its nearest neighbors. An intuitive way of visualizing this group of examples is in the form of a graph, such that (1) the examples are vertices, and (2) directed edges are drawn from the example for whom the search was performed to its nearest neighbors.\(^1\) The graph for \{Home, Memo\} is shown in Figure 7.2.

If we focus on the policy decisions of the vertices in Figure 7.2, we see that Bob’s decision for \{Home, Memo\} (i.e., allow) disagrees with the decision for both its nearest neighbors. This inconsistency or noise indicates one of two possibilities: (a) Bob made a mistake in labeling \{Home, Memo\} with the decision allow, or (b) Bob wanted to make a genuine exception for memos. Instead of making a guess, PyBE asks Bob. That is, we recommend Bob to label \{Home, Memo\} as deny for resolving this inconsistency, Bob may accept our recommendation.

\(^1\)This is a simplified explanation of the nearest neighbor graph to provide intuition; a more formal definition can be found in Section 7.3.3.
Figure 7.3 There is no majority policy decision among the nearest neighbors of \{Home, Photo\}.

or reject it and make an exception. Using such interactive recommendations, PyBE engages Bob in correcting potential errors.

Figure 7.3 shows the nearest neighbor graph for \{Home, Photo\}, and illustrates another type of inconsistency. In this case, there is no majority consensus among the neighbors of \{Home, Photo\}. A similar situation exists in the graph for \{Home, Document\}, which we do not show due to space constraints. If we look at the two graphs in Figure 7.2 and Figure 7.3, we realize that changing the policy decision of \{Home, Memo\} removes both the inconsistencies. Thus, PyBE capitalizes on the possibility that a few examples may cause the most noise, and recommends the user to change their labels. In our algorithm described in Section 7.3.3, we describe graph invariants to identify noise, and a greedy algorithm to find the optimal change. Section 7.5 demonstrates that our interactive approach finds five times as many errors as manual reviews by users.

Note that we do not claim to detect all errors, as the users’ examples may be completely consistent, but may still have errors. Instead, we recommend a best effort approach for engaging the user in detecting potential errors. More optimal solutions may be incorporated into PyBE in the future.

### 7.3 The PyBE Algorithm

This section describes our algorithm for predicting policy decisions, and the active learning approach. As stated previously, distance between policy examples is the distance between their scenarios. For simplicity, however, our discussions are in terms of the policy example.

Our policy examples (i.e., the scenarios) are Boolean functions over \(n\) variables (i.e., tags), denoted by \(\mathcal{B}_n\). However, we restrict our attention to functions that are conjunctions
of variables (e.g. \( x_1 \land x_2 \land x_3 \)). Such a function \( f \) can be represented as a set \( I(f) \subseteq \{1, 2, \cdots, n\} \) (e.g., if \( f = x_1 \land x_3 \land x_5 \), then \( I(f) = \{1, 3, 5\} \)). Our policy examples belong to this restricted class (denoted by \( \mathcal{P}_n \)).

We had two requirements for the learning-algorithm to infer policies: (I): non-parametric (does not rely on models with a fixed set of parameters). (II): easy explanation (easy to present to the user how the policy was inferred). For this reason we chose a variant of the \( k \) nearest neighbor (kNN) classifier [Mur12]. A kNN algorithm simply “looks at” the \( k \) points in the training set that are nearest to the test input \( x \), counts how many members of each class are in the set, and returns that empirical fraction as the estimate.

Recall that our goal is to label a policy example \( p \in \mathcal{P}_n \) with the decision 1 (i.e., allow) or 0 (i.e., deny). We are also given a set of examples along with known labels (i.e., policy decisions). Our algorithm is inspired by the kNN algorithm and works as follows: given a new policy example \( p \in \mathcal{P}_n \) with an unknown label, we find the set of \( k \) policy examples \( N(p) = \{p_1, \cdots, p_k\} \) closest to \( p \) according to the metric \( \mu \) (described in the next subsection) and then associate the label to \( p \) that corresponds to the majority labels of the policy examples in \( N(p) \). Our variant of kNN only considers examples at the closest distance for inclusion in \( N(p) \). We describe how to address situations where no majority exists in Section 7.3.4.

We use active learning to assist the user in correcting potential labeling errors in the user's policy examples. When we find that certain conditions are not true (e.g., the label of a policy example \( q \in \mathcal{P}_n \) is different from the majority label among its neighbors \( N(q) \)), we recommend a change in the label (e.g., change allow to deny). We now describe our metric \( \mu \), its weighted form \( \mu_w \), and the active learning phase.

### 7.3.1 The Metric

Let \( f \) and \( g \) be two Boolean functions over \( n \) variables \( x_1, x_2, \cdots, x_n \). A metric between \( f \) and \( g \) (denoted by \( \mu(f, g) \)) can be defined as follows:

\[
1 - \frac{\#(f \oplus g)}{2^n}
\]
Where $\oplus$ represents exclusive-or and $\#(h)$ is the number of satisfying assignments of the Boolean function $h$. Recall that computing the number of satisfying assignments of a Boolean function is a hard problem ($\oplus$-P complete [AB09]). However, for our special case (i.e. policies are conjunctions of variables) this metric is easy to compute. Next we describe the metrics for the functions in the set $\mathcal{P}_n$.

Consider two functions $f_1$ and $f_2$. Let $\{n\} = \{1, 2, \cdots, n\}$. Consider three sets of indices $I_{1,2}$ (indices of variables neither in $f_1$ nor $f_2$), $I_1^1$ (indices of variables in $f_1$ but not in $f_2$) and $I_2^1$ (indices of variables in $f_2$ but not in $f_1$); i.e., $I_{1,2} = \{n\} \setminus (I(f_1) \cup I(f_2))$, $I_1^1 = I(f_1) \setminus I(f_2)$, and $I_2^1 = I(f_2) \setminus I(f_1)$. An assignment $\sigma$ is a Boolean vector of size $n$ of the form $\langle b_1, b_2, \cdots, b_n \rangle$ and $f(\sigma)$ denotes the value of the function $f$ for assignment $\sigma$.

Consider an assignment $\sigma = \langle b_1, b_2, \cdots, b_n \rangle$ such that $f_1(\sigma) = 1$ and $f_2(\sigma) = 0$. Then for all $i \in I(f_1)$, $b_i = 1$, and there should be at least one $i \in I_1^2$ such that $b_i = 0$. For $i \in I_{1,2}$, $b_i$ can assume any value. Consider all the indices in $I_1^2$. There should be at least one $j \in I_2^2$, such that $b_j = 0$ (we want $f_2(\sigma) = 0$). Therefore, the number of satisfying assignments $\sigma$, such that $f_1(\sigma) = 1$ and $f_2(\sigma) = 0$ is

$$2^{k_2}(2^{k_2} - 1)$$

where $k_{1,2} = |I_{1,2}|$ and $k_2 = |I_2^2|$. Explanation for the formula is as follows: all variables with indices in the set $I_{1,2}$ can be given any value (resulting in the term $2^{k_{1,2}}$). All the variables with indices in $I_1^2$ can be given any values as long as one of them is 0, so an assignment where all variables with indices in $I_1^2$ is assigned 1 is excluded (this results in the term $2^{k_2} - 1$).

A symmetric argument shows that the number of satisfying assignments $\sigma$ such that $f_1(\sigma) = 0$ and $f_2(\sigma) = 1$ is

$$2^{k_{1,2}}(2^{k_1} - 1)$$

where $k_{1,2} = |I_{1,2}|$ and $k_1 = |I_1^1|$. Adding the two terms, we have that $\#(f_1 \oplus f_2)$ is $2^{k_{1,2}}(2^{k_1} + 2^{k_2} - 2)$. Therefore, the metric $\mu(f_1, f_2)$ in this case is:

$$1 - \frac{2^{k_{1,2}}(2^{k_1} + 2^{k_2} - 2)}{2^n}$$

where $k_{1,2} = |I_{1,2}|$, $k_1 = |I_1^1|$, and $k_2 = |I_2^1|$. Intuitively, $k_1$ is the number of variables that appear in $f_1$ but not in $f_2$, $k_2$ is the number of variables that appear in $f_2$ but not in $f_1$, and $k_{1,2}$ is the number of variables that appear in neither $f_1$ or $f_2$. Let $k = n - k_{1,2}$, which is the number
of variables that appear in \( f_1 \) and \( f_2 \) (i.e., \( k = |I(f_1) \cup I(f_2)| \)). The metric \( \mu(f_1, f_2) \) can be simplified as follows:

\[
\mu(f_1, f_2) = 1 - \frac{2^{k_1} + 2^{k_2} - 2}{2^k}
\]

Note that higher values of the metric indicate closeness. Further, recall Bob’s example from Section 7.2.2. In the unweighted case, the distance metric (i.e., \( \mu \)) score between \{Home, Document\} and \{Document\}, as well as between \{Home, Document\} and \{Home, Photo\} is 0.75.

### 7.3.2 The Weighted Metric

For security policies, some variables are more important than others; e.g., recall the Home tag from Bob’s policy in Section 7.2.2. To incorporate the importance of variables we introduce a weighted version of our metric. As before, we will consider a Boolean function over \( n \) variables \( x_1, x_2, \cdots, x_n \). However, in this case we have two weights \( w^0_i \) and \( w^1_i \) associated with each index \( 1 \leq i \leq n \). The weight associated with an assignment \( \sigma = \langle b_1, \cdots, b_n \rangle \) (denoted as \( w(\sigma) \)) is

\[
\sum_{i=1}^{n} w^0_i (1 - b_i) + w^1_i b_i.
\]

Given a set of Boolean assignments \( S \), define \( w(S) \) as \( \sum_{\sigma \in S} w(\sigma) \) – the sum of weights of all assignments in \( S \). Given a Boolean function \( f \), \( w(f) \) is the weight of the set of satisfying assignments of \( f \). Using a simple recursive argument, the weight of all \( 2^n \) assignments \( \{0,1\}^n \) is:

\[
\prod_{i=1}^{n} (w^0_i + w^1_i)
\]

Given \( n \) pair of weights \( (w^0_1, w^1_1), \cdots, (w^0_n, w^1_n) \), a weighted metric between two Boolean functions \( f \) and \( g \) (denoted as \( \mu_w(f, g) \)) is defined as follows:

\[
1 - \frac{w(f \oplus g)}{\prod_{i=1}^{n} (w^0_i + w^1_i)}
\]

Note that if for all \( i \) we have \( w^0_i = w^1_i = 1 \), we get the previous metric (i.e., the unweighted case).
As before, consider two Boolean functions $f_1$ and $f_2$ with index sets $I(f_1)$ and $I(f_2)$. Let the index sets $I_1$ and $I_2$ be as defined before. Define the following three quantities:

\[
\begin{align*}
  z_1 &= \prod_{i \in I_2} (w_i^0 + w_i^1) - \prod_{i \in I_1} w_i^0 \\
  z_2 &= \prod_{i \in I_1} (w_i^0 + w_i^1) - \prod_{i \in I_2} w_i^0 \\
  z &= \prod_{i \in I(f_1) \cup I(f_2)} (w_i^0 + w_i^1)
\end{align*}
\]

The metric $\mu_w(f_1, f_2)$ can be defined as:

\[
\mu_w(f_1, f_2) = 1 - \frac{z_1 + z_2}{z}
\]

The argument is exactly same as before. The reader can check that for the unweighted case (i.e. for all $i$ we have $w_i^0 = w_i^1 = 1$) we get the previous metric back.

**Setting weights:** Next we describe an algorithm to set weights. Given a set of variables $V = \{x_1, \cdots, x_n\}$, suppose we are given a partial order $\preceq$ on $V$ (e.g., $x_i \preceq x_j$ means that $x_j$ is more “important” than $x_i$). Next we construct a function $L : V \rightarrow [n]$ that assigns integers between 1 and $n$ to each variable in $V$ and has the property that $x_i \preceq x_j$ and $j \neq i$ implies that $L(x_i) > L(x_j)$.

We can assign higher weights $w_i^1$ to variables that have a lower value according to the function $L$ and set all the weights $w_i^0$ to 1. Note that it is not necessary to precisely define a mechanism for assigning weights, as long as the ordering imposed by $L$ is preserved.

### 7.3.3 Active Learning

In an ideal scenario, users would provide accurate examples to PyBE. However, as even expert users may not always be accurate [Yen14; Ion15], we expect a small margin of error in the policy decisions provided by the user; e.g., a typo resulting in 1 being accidentally marked as 0. We use active learning to find and correct potentially incorrect labels, by asking

---

\[\text{Such a function can be constructed by topologically sorting a directed graph whose nodes are } V \text{ and there is an edge from } x_j \text{ to } x_i (j \neq i) \text{ iff } x_i \preceq x_j.\]
users to relabel certain chosen examples. The approach of relabeling samples to remove errors has been shown to be effective even with non-experts by prior work [She08].

In our approach, the examples and their nearest neighbors are arranged as a graph, which allows us to relabel existing examples in a systematic manner if certain invariants on the graph are not true. In other words, the graph we are about to describe gives us a systematic way to evaluate the conditions that may be indicators of user error, and relabel examples during the active learning phase.

Let $G = (V, E, L_V, L_E)$ be a 4-tuple where $V \subseteq \mathcal{P}_n$ is the set of labeled policy examples, $E \subseteq V \times V$ is the set of edges, $L_V$ maps each vertex $v \in V$ with a label 1 (signifying allow) and 0 (signifying deny), and $L_E$ labels each edge $e \in E$ with a non-negative real value (i.e., $L_E(v, v')$ is $\mu(v, v')$, which is the distance between the examples $v$ and $v'$). The set of neighbors $N(v)$ of a vertex $v \in V$ is the set $\{v' | (v, v') \in E\}$ and intuitively represents all the nearest-neighbors of the policy $v$.

**(Inv-1): Majority label exists.** This invariant states that for all $v \in V$, its set of neighbors $N(v)$ have a majority label (i.e., more than $\frac{|N(v)|}{2}$ vertices in $N(v)$ have the same label $L_V(v)$).

**(Inv-2): Agreement with the majority label.** This invariant states that if invariant Inv-1 is true, then for every $v \in V$ its label $L_V(v)$ agrees with the majority label of its neighbors $N(v)$.

Intuitively we want the graph $G$ corresponding to our policies to satisfy invariants Inv-1 and Inv-2. If the graph $G$ violates either of the invariants, then we recommend relabeling of examples to the user to establish the two invariants.

Figure 7.4 shows instances of the graph for some vertex $p$ that violate the invariants. In Figure 7.4a, there is no majority label among $p$’s neighbors, which can be resolved by relabeling either $q$ or $r$. Further, in Figure 7.4b, the label on $p$ disagrees with the majority, which can be resolved by relabeling $p$.

We use a simple greedy approach to recommend changes, briefly described as follows: Consider a function $\mathcal{V}(G)$ that counts the total violations of both Inv-1 and Inv-2 in graph $G$. Further, consider a function $\mathcal{C}(v, G)$ that measures the impact of a potential label change on violations, i.e., returns the decrease in $\mathcal{V}(G)$ after a temporary change in the label of $v$ (i.e., $L_V(v)$). The label change that causes the maximum decrease in $\mathcal{V}(G)$ is optimal. Therefore, at each iteration, we find the optimal vertex, $v_{opt}$, by maximizing $\mathcal{C}(v, G)$ over all $v \in V \setminus V_{visited}$, where $V_{visited}$ is the set of all vertices that have been recommended to
Figure 7.4 NN graph for vertex \( p \), consisting of neighbors \( q \) and \( r \) at the closest distance \( d \), violates Inv-1 and Inv-2.

...the user previously. We add \( v_{opt} \) to \( V_{visited} \), and recommend the user to change \( L_V(v_{opt}) \). If the user accepts, we change \( L_V(v_{opt}) \). We reiterate until all the vertices are visited or until there are no more violations.

Alternately, invariants may also be satisfied by generating additional examples. As our current approach satisfies the objective of using noise for finding potentially incorrect policy decisions, we did not explore this alternative any further.

7.3.4 Addressing no majority during prediction

As described previously, we predict the label for a new example \( p \) as the majority label of its nearest neighbors \( N(p) \). In case there is no majority label, we use the following method to predict the label:

We eliminate the first neighbor that is not a mutual neighbor, i.e., if there is a labeled example \( q \) such that \( q \in N(p) \) but \( p \notin N(q) \), we remove \( q \) from \( N(p) \), thereby converging on a majority. In case such elimination is not possible, i.e., if all neighbors in \( N(p) \) are mutual neighbors, we deny by default. Our method considers the value of the distance between neighbors to resolve a tie, instead of randomly discarding one example (i.e., by only considering only an odd number of examples in \( N(p) \)). Our evaluation demonstrates that PyBE is better than a baseline of random guessing.

Finally, such cases of no majority were rarely seen during our evaluation described in Section 7.4; i.e., less than 6% test examples had no majority, and less than 3% were denied...
by default. However, as these results may be limited to our sample of study participants, we do not make any general claims.

7.4 Evaluation

We performed an IRB-approved feasibility study with expert users to evaluate the effectiveness of our approach. We chose experts under the hypothesis that they prefer more complex policies, the complexity of which makes them challenging to predict. Support for this hypothesis comes from the fact that non-expert users are more likely to employ binary security practices, (e.g., only visiting known sites rather than deciding based on security-related attributes like usage of https [Ion15]) and evidence that knowledge of security risks can increase sensitivity to security when making data decisions [Rad12]. Note that no personally identifiable information (PII) was collected from the participants. We plan to release a sanitized version of our data on publication.

The data collection and experiments were performed using semi-structured interviews. Most tasks (i.e., collecting examples, reviewing examples, and testing) took about 75 minutes, whereas collecting weights took about 20 minutes on average. While breaks were offered as a part of the experimental design, no participant elected to take their break, indicating that the cognitive load associated with this approach is allowable.

Our study is motivated by the following research questions:

**RQ1** How accurate are our predictions for random, unlabeled scenarios that may occur at runtime?

**RQ2** What are the causes for incorrect predictions?

**RQ3** Do users make mistakes in their examples?

**RQ4** Does our active learning approach help the user find mistakes in their examples?

We now describe the study setup, followed by the data collection and experiments. We discuss results in Section 7.5.
Table 7.3 The policy targets used in our study, along with the action they control.

<table>
<thead>
<tr>
<th>Policy Target</th>
<th>Action Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkCloud</td>
<td>Export of data to the enterprise cloud</td>
</tr>
<tr>
<td>PersonalCloud</td>
<td>Export of data to the personal cloud</td>
</tr>
<tr>
<td>WorkEmailApp</td>
<td>Export of data by the enterprise email application</td>
</tr>
<tr>
<td>PersonalEmailApp</td>
<td>Export of data by the personal email application</td>
</tr>
<tr>
<td>SocialApp</td>
<td>Export of data by the social network application (e.g., Facebook)</td>
</tr>
</tbody>
</table>

7.4.1 Study Setup

In this study, participants were asked to consider a smartphone environment, where personal and work data would be at risk of unauthorized exfiltration from the device. We now describe the participants (i.e., expert users), policy targets and information tags involved in the study.

7.4.1.1 Expert Users

We recruited 8 graduate student researchers from a security research lab for this study (denoted as $P_1 \rightarrow P_8$). All our participants had at least 1 academic year of experience (2.5 years on average) in security research at the time of this study, including at least one research project and two graduate-level courses in security or privacy. Additionally, our participants were actively researching topics in the areas of computer security and privacy at the time of the study. We use the security-focused course-work and research as an indicator of general security-awareness, and assume the participants to be well-aware of their own security and privacy requirements. Additionally, we confirmed that all our participants used their smartphones for both work and personal data. Finally, through an informal discussion of participant background knowledge, we confirmed that the participants were aware of the threat of exfiltration of work and personal data by third party applications on smartphones, as discussed by prior work (e.g., TaintDroid [Enc10]).

7.4.1.2 Policy Targets

Table 7.3 provides the policy targets (i.e., policies) used in our study. We used policy targets that were similar to the WorkCloud target discussed in Section 7.2. As described previously,
each target is treated as an independent policy. Thus, a policy target may also be called a policy for simplicity (e.g., the user’s WorkCloud policy). We considered 5 policy targets belonging to one of the following two categories: policies that (a) restrict the destination Web domain to which data can be exported or (b) restrict the exporters (i.e., applications) that are permitted to export data. That is, the WorkCloud and PersonalCloud policies regulate export of data to the work and personal cloud respectively. The WorkEmailApp, PersonalEmailApp and SocialApp policies regulate export by the user’s work email client, personal email client, and social network client (e.g., the Facebook app) respectively.

7.4.1.3 Information Tags

We provided users with 9 predefined secrecy tags, based on tags available in popular note-taking applications (e.g., Google Keep, Evernote). To enable our experts to create any complex policy they desired, we allowed them to create new tags as well. The tags (user-created or predefined) were primarily of two kinds, namely tags that defined the location or time at which the information was created (e.g., Work, Afterhours, MedicalFacility), or the type or class of information (e.g., Receipt, WhiteboardSnapshot). Appendix C.1 provides the tags used in this study.

7.4.2 Data Collection

This section describes the approach used for collecting the initial specification datasets and weights from participants.

7.4.2.1 Collecting Policy Examples

Participants were allowed to use our predefined tags, or create their own tags. Further, participants could combine tags into complex scenarios for creating examples. We placed no hard limit on the number of example scenarios each participant could provide. For each example scenario created with the tags, participants were required to provide policy decisions for the 5 policy targets (described previously in Table 7.3). Note that we collected

---

3We collected data for 7 policy targets, but discarded two policy targets before testing to limit user fatigue.
decisions for two more targets, but discarded them before testing to reduce user fatigue. The script for this task is provided in Appendix C.2.

A preliminary analysis of the examples collected from our participants led to two interesting observations, as follows:

Our participants created a total of 31 unique tags. One interesting observation was that out of the 40 total unique tags (i.e., 9 predefined and 31 provided by participants), 58% were specific to individual participants, while only 7 tags were commonly used by all the participants in their examples.

The observation regarding tags prompted us to perform a similar search for common examples across participants. We observed that out of the 246 example scenarios collected across participants, over 76% were specific to individual participants, and only 7 were common among all 8 participants. This demonstrates that relevant data-use scenarios may be unique to the individual, even among student researchers from the same research lab. This observation further motivates our research into user-specific policies for user-specific data.

Note that although we allowed our participants to create tags to get complex security policies, non-experts may not need to create tags under ordinary circumstances. We discuss ideas for collecting policy examples from non-experts in Chapter 8.

7.4.2.2 Obtaining Weights

As described before, we obtained 40 unique tags from the collected policy examples. On average, each participant used about 14 tags in their examples. We realized that suggesting numerical weights or ordering a large number of tags may be tiring for our participants. Therefore, we decided to categorize tags into semantic groups, and to allow participants to provide partial orders on groups instead of individual tags, thereby reducing the number of items a participant has to order. We could reduce our set of 40 tags into 9 mutually exclusive semantic groups, such that each tag would fall into at least one group. We performed two tasks with our participants for obtaining weights. Scripts for both tasks can be found in Appendix C.3.

First, participants customized our semantic grouping of tags to suit their understanding. The only restrictions on participants were that participants could not add or remove groups,
and that groups were to be mutually exclusive; i.e., a tag could only be a member of one 
group. An example grouping (for participant P1) is provided in Figure C.2 in Appendix C.3.

After ordering tags into groups, we provided participants with a partial order over 
the groups as an example. Participants could start from scratch, or customize the order 
we provided to generate their own custom order. The partial order relations specified 
by participants were used to generate weights as described in Section 7.3.2. Orders were 
collected in a spreadsheet, as seen from partial order created by P1 in Figure 7.5.

We confirmed each partial order relation by reading it out to the user. For example, for P1, 
we asked if “Personal data is more important than Work data”, to confirm Work;Personal 
shown in Figure 7.5.

Further, participants were informed that they could provide different partial orders 
for different policies, but most participants chose to keep the same general order for all 
policies. We describe the impact of this decision, and the need for an independent order 
per policy, in Section 7.6. Chapter 8 describes ideas on collecting weights from non-experts.

7.4.3 Experiments

7.4.3.1 Finding Errors Manually and with Active Learning

The review of examples was carried out 3 months after the initial specification, as most 
participants were unavailable over the summer break. We performed a two-step experiment 
to help participants identify and correct errors in their policy decisions. The scripts for 
these tasks can be found in Appendix C.4.

First, participants performed a manual review of their initial specification. Participants
reviewed a spreadsheet containing their policy examples, with one spreadsheet per policy target. Participants could update any policy decision they desired. For each update, participants were required to indicate a cause. We provided three hints for the cause, i.e., (1) correcting an error, (2) a change of mind, or (3) an inability to make a decision. Participants could also indicate a cause outside the three hints. Finally, participants indicated a justification for each change. The justifications provided valuable information for understanding the participant’s assumptions behind the change (e.g., “Work data is very confidential”). We used this information, along with justifications provided during testing, to understand the reasons behind inaccurate predictions made by PyBE (described in Section 7.6).

After the manual review, we performed a semi-automated review using our active learning approach (described in Section 7.3.3). For each participant, a separate review was performed for each policy target, i.e., we treated each participant-policy case as a separate policy specification problem. We used the changed examples from the manual review; hence, any errors discovered using this approach were in addition to those discovered during the manual review.

Our algorithm presented the participant with a series of suggestions (i.e., examples with corrected policy decisions, as shown in Figure 7.6). The participant could either accept or reject the suggestion. If the participant accepted the suggestion, we confirmed with the participant that the original decision was in error, and recorded it as an error identified and corrected by the algorithm. If the participant rejected the suggestion, we asked for a short justification to understand the participant’s policy preferences. To limit fatigue, we stopped at 15 suggestions for each participant-policy case.

### 7.4.3.2 Testing with Random Examples

For each participant, we randomly generated \( \frac{n}{2} \) new policy examples (i.e., new scenarios), where \( n \) was the number of examples initially provided by the participant. The random examples for a participant were created using the tags used in the participant’s initial exam-
The intuition is that the tags provided by the participant are relevant to the participant, hence random examples composed of them must be relevant as well. To mitigate labeling fatigue, the random examples included at most 3 tags. The experiment script is provided in Appendix C.5.

Participants provided the ground truth policy decisions for their testing examples, for each of the five policy targets. Apart from indicating “Allow” or “Deny”, participants were also provided the “I don't know” option to indicate their inability to understand the example, in which case we substituted the example with another random test example.

We predicted the policy decision for each test example using our algorithm, and computed accuracy for each participant-policy case. For incorrectly predicted examples, we asked the participant to confirm their decisions and provide short justifications. After concluding the testing experiment, we conducted a short informal interview to gain further insight into the participant’s decisions. The insights gained from the justifications and interviews helped us identify the problems described in Section 7.6.

### 7.5 Results

This section describes the results of our experiments, i.e., PyBE's accuracy in predicting policy decisions for new scenarios, and its effectiveness at helping our participants find and correct incorrect policy decisions in their policy examples. We start by describing the data collected during the initial policy specification (i.e., the specification dataset), and the random examples created for testing (i.e., the testing dataset).

**Specification dataset:** Table 7.4 shows the number of unique examples collected from our expert participants. The 8 participants provided 246 example scenarios in total, with policy decisions for 5 policy targets, resulting in a total of 1,230 initial labeled policy examples. The average number of tags used by participants for specifying their examples was 14, with 40 unique tags used across participants. Note that while we provided 9 predetermined tags, our participants created the remaining 31 tags. The tags used in our study are available in Appendix C.1.

**Testing dataset:** Table 7.5 shows the size of the testing dataset for each participant. Between 10 to 26 new random test scenarios were generated for each participant, based on
Table 7.4 Number of unique example scenarios created by each user, as well as the total number of policy examples created after assigning decisions for 5 policies.

<table>
<thead>
<tr>
<th>Users</th>
<th>Example scenarios created ( (n) )</th>
<th>Policy examples for 5 policies ( (n \times 5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>52</td>
<td>260</td>
</tr>
<tr>
<td>P2</td>
<td>23</td>
<td>115</td>
</tr>
<tr>
<td>P3</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>P4</td>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>P5</td>
<td>33</td>
<td>165</td>
</tr>
<tr>
<td>P6</td>
<td>26</td>
<td>130</td>
</tr>
<tr>
<td>P7</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>P8</td>
<td>42</td>
<td>210</td>
</tr>
<tr>
<td>Total</td>
<td>246</td>
<td>1,230</td>
</tr>
</tbody>
</table>

Table 7.5 Number of random policy scenarios created for testing predictions for each user. Policy decisions are acquired from the user and predicted by PyBE for 5 policies.

<table>
<thead>
<tr>
<th>Users</th>
<th>Random Test Scenarios ( (n) )</th>
<th>Labeled Test examples for 5 policies ( (n \times 5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>26</td>
<td>130</td>
</tr>
<tr>
<td>P2</td>
<td>11</td>
<td>55</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
<td>60</td>
</tr>
<tr>
<td>P4</td>
<td>12</td>
<td>60</td>
</tr>
<tr>
<td>P5</td>
<td>16</td>
<td>80</td>
</tr>
<tr>
<td>P6</td>
<td>14</td>
<td>70</td>
</tr>
<tr>
<td>P7</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>P8</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>Total</td>
<td>122</td>
<td>610</td>
</tr>
</tbody>
</table>

the number of initial examples they provided. Participants provided ground-truth policy decisions for 5 policies for each scenario, resulting in a total of 610 test examples.

### 7.5.1 Accuracy of Predictions

Our algorithm predicted decisions for all of the participants’ new (test) policy examples. The actual prediction time was negligible (i.e., less than 1 second for all the examples for one participant). Further, for less than 6% (36 out of 610) of our test examples we had no majority label (i.e., a tie). Applying the tie-breaker discussed in Section 7.3.4 resolved 19 of

---

4We performed testing with random samples instead of cross-validation, as the latter is generally performed to test "models", i.e., for supervised learning.
Table 7.6 Accuracy of PyBE in comparison with the *CoinFlip* (abbreviated to CF) baseline, for all 40 user-policy cases. Cases where the accuracy of CF is greater are highlighted in **bold**.

<table>
<thead>
<tr>
<th>Policy Target</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkCloud</td>
<td>0.96 0.50</td>
<td>0.73 0.48</td>
<td>0.66 0.49</td>
<td>0.83 0.49</td>
<td>0.56 0.51</td>
<td>0.93 0.50</td>
<td>0.90 0.49</td>
<td>0.67 0.49</td>
</tr>
<tr>
<td>PersonalCloud</td>
<td>0.77 0.50</td>
<td>0.55 0.5</td>
<td>1.00 0.50</td>
<td>0.75 0.52</td>
<td>0.75 0.51</td>
<td><strong>0.50 0.51</strong></td>
<td><strong>0.40 0.51</strong></td>
<td>0.71 0.49</td>
</tr>
<tr>
<td>WorkEmailApp</td>
<td>0.96 0.50</td>
<td>0.55 0.51</td>
<td>0.83 0.47</td>
<td>0.83 0.47</td>
<td>0.63 0.51</td>
<td>0.86 0.51</td>
<td>0.90 0.50</td>
<td>0.76 0.49</td>
</tr>
<tr>
<td>PersonalEmailApp</td>
<td>0.77 0.51</td>
<td>0.55 0.49</td>
<td>1.00 0.50</td>
<td>0.75 0.48</td>
<td>0.63 0.51</td>
<td><strong>0.50 0.51</strong></td>
<td>0.50 0.48</td>
<td>0.67 0.51</td>
</tr>
<tr>
<td>SocialApp</td>
<td>0.81 0.51</td>
<td>0.73 0.49</td>
<td>0.75 0.52</td>
<td>0.92 0.49</td>
<td>0.94 0.50</td>
<td>1.00 0.51</td>
<td>1.00 0.52</td>
<td>0.91 0.50</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.85 0.50</td>
<td>0.62 0.49</td>
<td>0.85 0.50</td>
<td>0.82 0.49</td>
<td>0.70 0.51</td>
<td>0.76 0.51</td>
<td>0.74 0.50</td>
<td>0.74 0.50</td>
</tr>
</tbody>
</table>

*The *CoinFlip* baseline values shown are the mean of 50 executions with a 95% confidence interval less than 0.03.*

Table 7.7 Accuracy of PyBE in comparison with the *MostFreq* (abbreviated to MF) approach, for all 40 user-policy cases. Cases where the accuracy of MF is greater are highlighted in **bold**.

<table>
<thead>
<tr>
<th>Policy Target</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkCloud</td>
<td>0.96 0.85</td>
<td>0.73 0.63</td>
<td>0.66 0.17</td>
<td>0.83 0.75</td>
<td><strong>0.56 0.88</strong></td>
<td>0.93 0.71</td>
<td>0.90 0.70</td>
<td>0.67 0.43</td>
</tr>
<tr>
<td>PersonalCloud</td>
<td>0.77 0.69</td>
<td><strong>0.55 0.91</strong></td>
<td>1.00 1.00</td>
<td>0.75 0.75</td>
<td>0.75 0.75</td>
<td>0.50 0.21</td>
<td>0.40 0.20</td>
<td>0.71 0.71</td>
</tr>
<tr>
<td>WorkEmailApp</td>
<td>0.96 0.85</td>
<td><strong>0.55 0.64</strong></td>
<td>0.83 0.17</td>
<td>0.83 0.75</td>
<td><strong>0.63 0.81</strong></td>
<td>0.86 0.71</td>
<td>0.90 0.80</td>
<td>0.76 0.48</td>
</tr>
<tr>
<td>PersonalEmailApp</td>
<td>0.77 0.69</td>
<td><strong>0.55 0.91</strong></td>
<td>1.00 1.00</td>
<td>0.75 0.75</td>
<td><strong>0.63 0.81</strong></td>
<td><strong>0.50 0.79</strong></td>
<td><strong>0.50 0.80</strong></td>
<td><strong>0.67 0.71</strong></td>
</tr>
<tr>
<td>SocialApp</td>
<td>0.81 0.73</td>
<td>0.73 0.46</td>
<td><strong>0.75 0.92</strong></td>
<td>0.92 0.58</td>
<td><strong>0.94 1.00</strong></td>
<td>1.00 1.00</td>
<td>1.00 1.00</td>
<td>0.91 0.91</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.85 0.76</td>
<td><strong>0.62 0.71</strong></td>
<td>0.85 0.65</td>
<td>0.82 0.72</td>
<td><strong>0.70 0.85</strong></td>
<td>0.76 0.68</td>
<td>0.74 0.70</td>
<td>0.74 0.65</td>
</tr>
</tbody>
</table>
these ties, while the rest (i.e., 3% or 17 out of 610) were denied by default. We now discuss
the accuracy of PyBE’s predictions.

On comparing our predicted decisions with ground-truth decisions provided by partici-
pants, we observe that PyBE predicts policy decisions with an average accuracy of over 76%
across all participants (RQ1). When analyzing the accuracy, it is important to note that each
participant-policy combination is treated as an independent policy specification problem,
and hence forms a separate test case. We first define a baseline and naive approach against
which we evaluate PyBE’s accuracy.

1. **The CoinFlip baseline:** The *CoinFlip* baseline provides the measure of accuracy of ran-
dom guessing, with an equal probability of a 0/1 outcome on each flip.

2. **The MostFreq naive approach:** We define *MostFreq* as an approach that predicts the
most frequent or majority policy decision from the initial examples, for the respective
participant-policy problem. For example, if a participant generally allows in policy decisions
for a certain policy target, *MostFreq* will predict allow for all new test examples for that
participant-policy problem. The insight behind *MostFreq* is that a naive learner is likely to
pick the majority class, in order to benefit from the consistent trend in the participant’s
policy decisions.

Table 7.6 shows the comparison of PyBE’s accuracy with *CoinFlip*, for each of the 40
participant-policy cases (i.e., 8 participants and 5 policy targets). PyBE performs better
than *CoinFlip* in all but 3 cases (i.e., 92% of the time). PyBE’s average accuracy of over 76%
is better than *CoinFlip* (50% accuracy). Additionally, PyBE performs better than *CoinFlip*
for all 8 participants, across all policies.

Table 7.7 shows a comparison between the performance of PyBE and the naive approach
*MostFreq*. PyBE performs better than *MostFreq* in 29 out of 40 (i.e., 72.5%) participant-policy
cases, and for 75% of the participants. The average accuracy of PyBE exceeds *MostFreq*
(71% accuracy) as well.

Note that *MostFreq*’s accuracy is abysmally low (e.g., 17%) in certain cases is due to its
over-dependence on the probability distribution of the training examples. That is, *Most-
Freq* only does well when the testing dataset consists of examples that are very similar to
the training set as a whole. PyBE does not make any assumptions about the probability
distribution of data, and hence does not suffer from the same flaw. We discuss the causes
The number of errors identified via manual review of examples (abbreviated as MR), and the additional errors found in the PyBE-assisted review. Cases where the manual review finds more errors are highlighted in **bold**.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkCloud</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>PersonalCloud</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td><strong>1</strong></td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>WorkEmailApp</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PersonalEmailApp</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SocialApp</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Of incorrect predictions (**RQ2**) in Section 7.6.

### 7.5.2 Effectiveness of Active Learning

We now discuss the effectiveness of PyBE’s active learning approach over manual reviews by participants, for finding labeling errors (i.e., errors in the policy decisions of examples) in their specification datasets.

Table 7.8 shows the number of labeling errors found by the participant through the manual review, followed by the additional errors found using PyBE’s interactive approach. The errors found using PyBE’s approach are additional as we used the corrected dataset from the manual review for the PyBE-assisted review, as described in Section 7.4.3.

First, we observe at least one labeling error in most participant-policy cases, i.e., in 30 out of 40, or 75% cases. Out of 1,230 total examples in the specification dataset, we observe 96 total errors (i.e., about 7.8%) (**RQ3**). While participants identify some errors manually, PyBE’s semi-automated process helps the participant identify and correct the maximum number of errors (80 out of 96, or about 83%). For all 8 participants, the total errors (across policies) found by PyBE are equal to or more than the participant’s manual review. Thus, we conclude that while participants make some error (average 7.8%) (**RQ3**), participants find more errors using PyBE’s interactive approach than by themselves (**RQ4**).

We note that P1 did not find any errors manually, nor did they agree to PyBE’s recommendations as they had confidence in their examples. Further, our accuracy is also the highest for P1 as seen in Table 7.6. However, given the absence of any trend in other cases,
Table 7.9 Breakdown of incorrect predictions into misconfigured weights (W), policy change (C), unconfirmed policy change (U) and label confusion (L), across all participants and policies.

<table>
<thead>
<tr>
<th>Policy</th>
<th>P1 W U L</th>
<th>P2 W U L</th>
<th>P3 W U L</th>
<th>P4 W U L</th>
<th>P5 W U L</th>
<th>P6 W U L</th>
<th>P7 W U L</th>
<th>P8 W U L</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkCloud</td>
<td>1 0 0 0</td>
<td>2 1 0 0</td>
<td>0 1 3 0</td>
<td>2 0 0 0</td>
<td>6 0 0 1</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 3 3 1</td>
</tr>
<tr>
<td>PersonalCloud</td>
<td>6 0 0 0</td>
<td>2 0 2 1</td>
<td>0 0 0 0</td>
<td>3 0 0 0</td>
<td>3 0 0 1</td>
<td>7 0 0 0</td>
<td>6 0 0 2</td>
<td>2 0 1 3</td>
</tr>
<tr>
<td>WorkEmailApp</td>
<td>1 0 0 0</td>
<td>2 2 1 0</td>
<td>0 0 2 0</td>
<td>2 0 0 0</td>
<td>5 0 0 1</td>
<td>0 2 0 0</td>
<td>0 1 0 0</td>
<td>1 2 0 0</td>
</tr>
<tr>
<td>PersonalEmailApp</td>
<td>6 0 0 0</td>
<td>2 0 2 1</td>
<td>0 0 0 0</td>
<td>3 0 0 0</td>
<td>4 1 0 1</td>
<td>7 0 0 0</td>
<td>5 0 0 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>SocialApp</td>
<td>4 1 0 0</td>
<td>2 0 0 1</td>
<td>3 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 2 0 0</td>
</tr>
<tr>
<td>Total</td>
<td>18 1 0 0</td>
<td>10 3 5 3</td>
<td>3 1 5 0</td>
<td>11 0 0 0</td>
<td>19 1 0 4</td>
<td>14 3 0 0</td>
<td>0 13 0 0</td>
<td>4 8 5 5</td>
</tr>
</tbody>
</table>

*Five incorrect predictions for P8 (i.e., 3 in PersonalEmailApp and 2 in WorkEmailApp) are not included in the table due to insufficient information.

we refrain from claiming any relation between the user errors found in the reviews and accuracy.

7.6 Analysis of Results

The approach described in this chapter is the first step towards our goal of a policy assistant. As a result, our objective is not only to demonstrate a feasible approach, but also to learn lessons for future work. With this motivation, we performed an in-depth study of our results to identify the general causes of incorrect predictions.

We manually analyzed each of the 141 incorrectly predicted test examples. For our analysis, we considered the following information collected during our study: (1) the justifications provided by the participants for their policy decisions, (2) the example(s) from the specification dataset found nearest to the test example by our algorithm, (3) the weights set on each of the tags involved in the test example, and (4) all the examples in the specification dataset that contain one or more of the tags included in the test example. The rest of this section describes the four causes of incorrect predictions that we identified. A detailed breakdown of the causes across participants and policies is provided in Table 7.9.
We found that a majority of our incorrect predictions (79 out of 141, or over 56%) were caused because the weights set by the participants contradicted their actual security preference. We confirmed our findings using justifications from participants that clearly indicated the tag or security preference that influenced their policy decision for a test example.

For instance, consider an incorrect prediction for $P_1$ for the PersonalCloud policy, where PyBE predicted the policy decision allow for $P_1$’s test example $\{\text{WhiteboardSnapshot, Work, ScannedDocument}\}$. However, the user provided the ground-truth decision of deny, and with the quote “no work data to personal cloud”. That is, the tag Work was confidential and hence important to $P_1$ with respect to the PersonalCloud policy target.

Table 7.10 shows a subset of the initial examples provided by $P_1$. Observe that all of $P_1$’s examples that contain Work (except one) deny export to the PersonalCloud. This leads us to conclude that $P_1$ has consistently thought of Work as important with respect to the PersonalCloud. However, this importance was not reflected in the weights, i.e., $P_1$ mistakenly assigned Work data a lower weight by ordering it lower than personal data, as we saw previously in Figure 7.5.

We investigated further by calculating numerical weights for each tag involved in this incorrect prediction, using $P_1$’s weight orders with the method described in Section 7.3.2. The tag Work had a low weight (i.e., 2), compared to other personal tags (e.g., MedicalFacility, weight 4). Therefore, rather than being correctly identified as close to
the work-related scanned document scenario specified by P1 (i.e., Work, ScannedDocument),
the test example was incorrectly identified as being close to the personal policy examples
{MedicalFacility,ScannedDocument} and {WhiteboardSnapshot,MedicalFacility},
both of which allowed export to the PersonalCloud. Hence, the decision was incorrectly
predicted as allow.

On raising the weight of Work to 5 (i.e., above personal tags), the test example was
correctly found closer to {Work,ScannedDocument}, resulting in a correct prediction. Note
that this increase in weight is not arbitrary, but guided by evidence of the user’s security
preferences. On correcting all misconfigured weights, we manually confirmed that our
overall accuracy rose to 89%. This includes predictions for P2 and P5 for whom PyBE had
the lowest accuracy, i.e., most incorrect predictions for these users are due to incorrect
weights (which can be fixed), as shown in Table 7.9.

Since misconfigured weights caused the maximum incorrect predictions (79 out of 141,
or 56%), we studied our data to find causes of incorrect weights. We made two interesting
observations, described as follows:

Observation 1: Inaccurate predictions resulted from participants only considering privacy
preferences when setting weights. We observed that a large number of inaccuracies occurred
when participants did not take their security preferences for specific security policy targets
(e.g., PersonalCloud) into account while setting weights, and set weights only based on
their general privacy preferences. However, since participants labeled examples based on
their security preferences, this resulted in incorrect predictions.

Recall that our participants were provided with the option of setting different weight-
group orders for different policy targets in Section 7.4.2. However, all participants (except
P8) set only a general order for all 5 policy targets, which only accounted for the their
privacy preferences. As a result, higher-weighted personal tags (e.g., MedicalFacility,
Home) had more influence on the policy decision, irrespective of the actual policy target. This
phenomenon caused false negatives for personal policy targets such as PersonalCloud and
PersonalEmailApp, since PyBE was forced to identify work-related test examples as closer
to personal examples, and hence predicted allow instead of deny (e.g., as seen in P1’s case
previously). Note that the number of false positives for the WorkCloud and WorkEmailApp
policies was lower, as participants generally denied export for random examples.
We confirmed that at least 26 incorrect predictions (out of 79 due to weights) were false negatives in predicting the PersonalCloud and PersonalEmailApp targets, which were directly caused by the participant considering privacy preferences while setting weights, but security preference (for work data) while labeling examples.

Observation 2: “Important” may not just mean confidential. In at least 14 of the test-examples incorrectly predicted due to weights, participants wanted to set a high weight for a non-confidential tag. From the justifications given by our participants during testing, we realized that participants intended to declassify data if a certain tag were present in the example.

This was in complete contrast with the initial understanding of the participants while setting weights, i.e., that confidential tags would have high weights. Thus, applying PyBE may require careful consideration of what is “important”, depending on the security goals of the policy. We discuss the lesson learned from this observation in Section 7.7.

7.6.2 Policy Change

A significant minority (30 out of 141, or over 21%) of our incorrect predictions resulted from a change in the participants’ policies, i.e., when participants explicitly disagreed with an earlier assumption they made about their policy.

Consider the example of P8, whose policy has changed with respect to the tag School. During the initial policy specification, the participant was employed off-campus, and hence assumed the tags School and Work to have different meanings. Therefore, the policy decisions for school were often different than for work. Between then and the testing phase, the participant changed jobs, and is now employed with the school. As a result, during testing, the participant provided similar policy decisions for work and school, and justified with the quote “school is work”. P8 admitted to this change during the post-testing interview. In all such cases, we confirmed the policy change with evidence provided by participants in their interviews or justifications.

Recall that the participants were provided with an opportunity to indicate a change of mind during the manual review of examples described in Section 7.4.3.1. All participants changed 35 decisions in total across policies. However, it is clear that participants were unable to find all of their potential changes with a manual review. Thus, a policy assistant may need to account for the participant’s change of policy and prepare in advance, as we
7.6.3 Unconfirmed Policy Change

For a small number of incorrect predictions (15 out of 141, or about 11%), we observed a clear contradiction between the participant's examples during specification and testing, but could not get a confirmation from the participant.

Consider P8's test example \{SavedToDevice, Audio\}, where the ground truth label allows export for the WorkCloud policy. However, all but one of P8's initially specified examples containing SavedToDevice or Audio deny export to the work cloud. P8's justification stated that the participant had “some level of trust in the work cloud”, which neither confirms nor denies a change of policy. Hence, we classify the cause for such cases as an unconfirmed policy change.

7.6.4 Tag Confusion

The least number of errors (i.e., 12 out of 141, or about 8.5%) were caused due to the ambiguity of some tags used in our study. The location or time-based tags (e.g., Home, Work, Afterhours) were intended to indicate the location or time of creation of different classes of data. As we did not place strict constraints on our expert participants, our participants also created examples where such tags could be used on their own (i.e., an example \{Home\} would mean data of an unknown type created at home). Justifications indicated that while participants had no trouble understanding the examples they had created in their specifications, some random test examples caused confusion. Less than half of our participants faced this problem, and for a few random text examples.

For example, \{Afterhours, Audio, Document\} could mean Audio created after hours, and added to a document whose origin is unknown, or a document created after hours, and added to an audio recording whose origin is unknown, and so on. Participants indicated such confusion for specific examples in their justifications during interviews. We describe a simple solution to this problem in Section 7.7.

Finally, we exclude five incorrectly predicted examples from P8's test dataset from our categorization. For the examples (3 for PersonalEmailApp and 2 for WorkEmailApp), the
participant suggested an inability to decide unless they knew the identity of the email receiver, which gave us no information. We did not face this situation with any other user or example.

### 7.7 Lessons

As seen in Section 7.6, many incorrect predictions of PyBE can be resolved by making a few simple but significant improvements (e.g., setting weights according to security preferences) to the application of PyBE. In this section, we describe lessons we learned from our study with expert users. These lessons address aspects of correctly using PyBE in practice, and also motivate problems for future work.

**Lesson 1:** Weight assignment should reflect security, and not just general privacy preferences. As seen in our study, the policy (target) may be semantically related to the tags (e.g., work data is more confidential with respect to the *PersonalCloud*). A generic weight assignment for multiple policies is bound to be inaccurate. Thus, weights should be declared independently for each policy, to reflect the policy-specific security preferences of the user. The usability challenge of doing so in a way that is not burdensome to the user is discussed in Chapter 8.

**Lesson 2:** Addressing the problem of “potential” change in the user’s policy is imperative. As we discovered in our study (described in Section 7.6), the user may change some or all of their security policy goals, but never inform the policy assistant. Thus, adapting to potential change in the user’s policy is a major requirement, and a challenge for future work in this area. We elaborate in Chapter 8.

**Lesson 3:** There might be more than one notion of “importance” depending on the security goals. During our study, we observed instances where participants wanted to assign more importance to non-confidential data. As our study was conducted with strong security requirements in mind, we did not account for such cases. Incorporating such a requirement into PyBE is feasible, although the numeric limits on weights of the confidential and non-confidential tags must reflect the security goals; i.e., if security is a priority, then the maximum possible weight for the most confidential tag should be higher than that for the most non-confidential tag. If usability is a priority, then the most non-confidential tag may
Lesson 4: Tag semantics should be carefully considered while applying PyBE. A small number of our incorrect predictions were due to ambiguity associated with specific tags as described in Section 7.6. That is, tags were defined such that \{Work, Photo\} could mean photo created at work, as well as some work data and an unrelated photo combined. One simple solution could be to use a separate representation to denote data created at a particular location or time. For example, a photo created at work may have the tag WorkPhoto, while a derived data object containing unidentified work data and a photo from an unknown location forms \{Work, Photo\}. This solution and its implications will be explored in the future.

7.8 Threats to Validity

In this chapter, we provide a general framework for specifying policies for user-specific data. Individual aspects of our framework may be iteratively refined in the future. We identify specific limitations of the current state of our approach and its evaluation as follows:

We evaluate feasibility with expert users. While our participants provide a significant number of policy examples, the absolute number of participants is small and not diverse, hence we cannot generalize to the broader user population. However, because this specialized set of users are likely to have more complex policies than most users, we view our feasibility study as a sufficient “stress test” of PyBE.

Additionally, since even expert users can make bad security decisions [Yen14], our expert-specified policy examples are not expected to be error-free. Indeed, we use our interactive approach to help users find potential errors.

Our policy example (scenario) is described as a conjunction of variables (Section 7.2.1). While it is easy to see how such a format may generalize to any policy that may be expressed as a conjunction of data objects or conditions, a thorough evaluation of expressibility may be required.

Finally, in Section 7.3.3, we propose a simple greedy approach to satisfy graph invariants. A more complex approach (e.g., using dynamic programming) may be integrated into PyBE without any significant changes.
7.9 Summary

We introduced the paradigm of Policy by Example (PyBE) for user-specific policy specification. PyBE enables users to express data-use scenarios in policy examples, and predicts policy decisions for new scenarios. A salient aspect of PyBE is its active learning approach for engaging users in finding and potentially incorrect policy decisions in their examples. In our feasibility study with expert users, PyBE demonstrated over 76% prediction accuracy, and better average performance than a baseline and a naive approach. A significant finding was that PyBE’s interactive approach was five times as effective as a manual review in finding errors. Finally, we analyzed our incorrect predictions and learned lessons that motivate future research directions in this promising new domain.
Third party applications provide for a large fraction of the user’s computing needs on modern commodity platforms, by allowing the user to create, process, manage and share information. Prior work has demonstrated that applications cannot be implicitly trusted, and often leak private user data [Enc11; Enc10]. Hence, it becomes necessary to treat untrusted third party applications with caution.

Protecting user resources from applications has been the focal point of smartphone OS security research for the last decade. However, prior work has been limited in its recognition of the problem space, focusing only on the existing, well-defined types of user information (e.g., IMEI, Location). This dissertation challenges this view, by bringing attention to the security requirements of application-specific user data (e.g., email attachments, documents).
As discussed in previous chapters, providing fine-grained security policy and enforcement for application-specific user data is hard, as its security context is only known to the user or the application that creates or modifies the data object. The techniques described in this dissertation enable the protection of application-specific data in modern commodity platforms. For instance, Aquifer [NE13] facilitates decentralized policy specification on Android, allowing applications to specify the secrecy policies for their own data. Weir [Nad16] presents novel enforcement primitives that make DIFC enforcement practical as well as secure on Android. Finally PyBE provides an approach for specifying user-specific security policies for abstract user data. Lessons learned from this dissertation reveal the need to consider user-directed sharing as an unavoidable form of data use for protecting user data, and motivate future research in policy specification and security challenges in emerging commodity platforms (e.g., Internet of Things, i.e., IoT).

8.1 The Inevitability of User-directed Sharing

Applications on modern commodity platforms have evolved to be purpose-specific. Thus, modern commodity platforms have developed mechanisms that allow users to combine the abilities of multiple applications. While this trait is most prominent in Android and Windows Phone, even iOS, which isolated applications for most of its history, has recently moved to this model of user-directed inter-application communication and sharing.

User-directed application mashups are becoming a central feature of emerging computing paradigms such as the Internet of Things, where multiple physical and virtual applications must interact to fulfill the user's vision. Strong evidence of this phenomena can be observed in the emergence of trigger-action programming services (e.g., IFTTT [TT10], Stringify [Str16]) as well as adoption of dynamic user-defined application pairings in popular smart home systems (e.g., Samsung SmartThings [Sam]). Application interoperability and data sharing is a necessity for commodity platforms, and user-directed sharing facilitates keeping the user in the loop.

The security community needs to recognize user-directed data sharing between applications as an indispensable and permanent fixture of modern commodity platforms, and to treat its availability as a functional requirement for any user data security solution. Present
proposals that place significant restrictions on user-directed sharing, specifically the ones that rely on coarse-grained containerization (e.g., Samsung Knox [Sam13] and Android for Work [Anda]) hinder application interoperability, and are ill-equipped to address data secrecy threats within the container. Providing data security without denying the reality of user-directed data sharing requires security research to switch gears from coarse-grained containers to information flow control (IFC). While Chapters 6 and 7 demonstrate feasible IFC policy and enforcement in this setting, they expose new challenges in data management and usability that may require additional research effort.

8.2 Future Directions in User-driven Policy Specification

As systems grow in complexity and rely on users to make policy decisions, technology must evolve to engage and assist the user in this process. As a result, user-driven policy specification is a irreplaceable requirement for deploying security systems that protect user resources. The Policy by Example (PyBE) paradigm proposed Chapter 7 lays the groundwork for user-driven policy specification, and demonstrates feasibility of an example-based approach. In this section, we identify four elemental research questions for future work in this area, namely (1) What policy information does the system need from the user?, (2) How to acquire policy information from non-experts, (3) What contextual information does the user need to provide policies?, and (4) How to account for a change in the user’s policies?. The remainder of this section describes these open questions and the associated future work.

8.2.1 Quantifying the information needed from the user

The security policy of a system can be acquired using two complementary ways: internally, i.e., by statically encoding the policy in applications or the system, or externally, i.e., from a policy oracle such as the user. Statically encoding the security policy in the program requires no effort on the part of the user, but may lack precision. For instance, modern commodity platforms for smartphones or IoT devices often protect all photographs taken via the camera using the same policy, which results in similar enforcement for photographs with disparate security contexts, e.g., a photograph of a person may be more privacy-sensitive than that
of a random object. On the contrary, acquiring the entire security policy from the user by querying on every security-sensitive event may be precise, but is impractical and insecure. Precisely partitioning the security policy into user-defined and system-defined components is a problem of critical importance for security systems.

A practical system must adopt a balanced approach, of combining the security-relevant information (i.e., the context) available with the user and the static context available with the system. In the interest of both usability and security, future work should determine the minimum questions that must be asked to the user, in order to provide sufficient context to the security system for accurate and precise access control enforcement. Achieving this objective involves two research tasks. First, future work should model access control policies and design a metric that quantifies access control information, in order to determine how much information is required by the system for sound and precise enforcement, and what fraction of it must come from the user. Second, future work should identify the minimum set of questions that the user must answer to provide the required context, which may use or extend querying strategies explored in prior work in active learning [LG94; Seu92; RM01].

8.2.2 Acquiring policies from non-experts

Chapter 7 demonstrate the feasibility of the Policy by Example (PyBE) approach with expert users. Measuring and improving the usability of PyBE for non-experts is a natural direction for future research. We now describe certain aspects of PyBE that may be revisited or enhanced for non-experts:

Suitability of Tags: Prior work demonstrates that users are capable of reasoning about access control using organizational tags [Kle12]. However, Chapter 7 shows that users may experience confusion if certain tags are combined, although such cases were rarely observed. The suitability of tags for describing security-sensitive scenarios requires further evaluation.

Collecting examples: To ease the cognitive load involved in creating tags, non-experts may be provided with a large and diverse collection of tags (e.g., the 40 tags obtained in our study) as a baseline for specifying examples. Providing initial tags may also motivate the user to create new tags.

However, non-experts cannot be expected to specify examples without a user interface.
A challenge for future work is to acquire policy examples without burdening the user or hampering the user experience. One possible alternative for future work is to consider the approach of using “interactive dropdowns” to collect examples, as described in prior work on usable policy authoring [Joh10b; Joh10a].

**Collecting Weights:** Collecting weights from non-experts while reducing user-burden is another challenge for future work. Our approach for setting weights (described in Section 7.3.2) allows users to describe relative tag weights in terms of partial order relations on tags. To improve the usability of this approach we are considering using visual “sliders” for weight collection, which will allow users to easily select more precise weights.

Additionally, incorporating PyBE into an existing system may require careful consideration of run-time overrides. That is, power-users may want to override policy decisions predicted by PyBE during runtime, or provide feedback. The advantage of incorporating such overriding mechanism is that the user’s correct decision may be added to the user’s policy examples to improve future predictions. A challenge for future work is to indicate to the user that a prediction has been made, while causing minimum interference with the user’s task.

### 8.2.3 Identifying and providing the contextual information needed by the user

A security decision cannot be made by an external observer lacking the necessary context [Bar06]. As evidenced in Chapter 7, even expert users may be unable to make security decisions without the necessary context. To assist the user in making an accurate security decision, the security framework should provide the user with the complete context of the access that requires the decision. For instance, apart from the subject and object identities, the user may also need to know details of the object’s provenance, or the subject’s prior and projected activities. For user-specific data-use scenarios, the degree of context required may vary with the user (i.e., P8 in Chapter 7 exhibits confusion for the email policy, while other participants do not). Additional research may be needed to identify user-specific contextual criteria, and effective methods of presenting the user with the context.

Further, simply providing all the necessary information (i.e., the context) may not be sufficient. In fact, prior work has shown that users may get overwhelmed with information, and
exhibit signs of confusion even when addressing simple permission requests [Fel12]. Therefore, the security framework must provide the user with an abridged version of the context, or what is colloquially known as the “tldr” form. Similar efforts at conveying information have been explored in non-security domains; e.g., “tldr-pages” uses a community-driven approach to provide short but meaningful alternatives to traditional Unix/Linux man pages, making the information accessible to non-experts [TLD]. However, as data as well as data-use scenarios may be user specific, leveraging community support may be challenging.

8.2.4 Accounting for change in the user's policy

Our preliminary study in Section 7.6 shows that over 21% incorrect predictions were due to change in the participant’s policy. Future research may be directed at solving the problem of detecting potential change in policy from current examples. Our intuition is that while detecting change in some cases may be impossible without external input (e.g., change in policy for Work data due to loss of employment), there is potential in evaluating solutions to this problem for other cases. We describe three challenges in this area, as follows:

1. **Identifying the causes of policy change:** Identifying and studying the causes of change in security policy is a precondition for demarcating the line between policy changes that can and cannot be addressed with machine learning. A longitudinal user study may need to be performed to find if users change policies frequently, and to identify the causes, as prior work has done for file access control policies [SG09].

2. **Identifying the time of policy change:** Identifying when a policy change happens is hard without any input from the user. One approach to address this challenge is to encourage users to report policy changes. Future work may take lessons from persuasive technologies that change user behavior and motivate users to participate in the desired activity [Con09; MC12; Pol10].

3. **Identifying the example(s) to change:** Instead of waiting for the user to indicate change, the learner may be able to use existing information (e.g., tag weights, frequency in examples) to detect what example or tag may be subject to change. Our intuition is that strategies used for cache replacement (e.g., least recently used or LRU [SJ94]) may be applicable to this domain, at least as a starting point.
8.3 Security of User Resources in Emerging Environments

The security architectures of commodity operating systems evolve over time, often adapting lessons learned from security research. The treatment of applications as security principals in modern commodity operating systems (e.g., Android, iOS) and even emerging commodity platforms such as smart homes (e.g., Samsung SmartThings [Sam], Apple HomeKit [App]) provide the most decisive examples of this phenomenon. However, OSes may also retain some of the drawbacks of their predecessors. For instance, findings of recent studies demonstrate that the coarse-grained access control frameworks of smart homes display limitations that were also previously seen in Android and other mobile platforms [Fer16b; Fer16a].

Further, IoT platforms, and specifically smart homes, demonstrate a form of user-directed sharing that is similar to what is observed on modern OSes such as Android. For example, on SmartThings, applications or “SmartApps” often request functionality that may be offered by more than one physical device, virtualized as a “Smart Device”. The user chooses the device that provides the functionality, in effect directing the functionality between two security principals. This mechanism is in principle to the intent messaging and intent-filter mechanism deployed in Android, where the user chooses the target of an intent message. Additionally, user-directed chaining of smart home applications is made easy with services such as Stringify [Str16] and IFTTT [TT10].

The presence of user-directed sharing between security principals exposes user resources in smart homes and other emerging platforms to confidentiality and integrity threats. The lessons learned from this dissertation may be used to design transitive security guarantees for such systems. However, the unique security and functionality-related challenges of such platforms may require novel solutions. The rest of this section describes three such challenges.

8.3.1 Automation, and the Absence of UI Interaction

IoT platforms are driven by the need to automate. At present, users can program entire tasks involving multiple applications in their homes (e.g., routines in Apple HomeKit [App]), and schedule the tasks to execute at predefined intervals. As a result, the system may only notify the user of a security-relevant event, but may not expect a synchronous security
decision from the user. This setup forces the user to agree to all security-relevant events while installing the application. For instance, in the case of a data secrecy to smart homes (e.g., FlowFence [Fer16a]), application must declare in code the secret data they have to offer, the secret data they intend to consume, and the flows involving secret data. These declarations are offered to the user at install-time, who may allow or deny the flows involving secret data. However, such static declarations provide only coarse-grained enforcement. If security classes for user resources are defined statically based on type, the user may not be able to define precise security classes using the user's context at runtime. For example, all camera data may not belong to the same security class, and photos taken at different times or of different objects may have disparate security requirements, only known to the user. Policy prediction at runtime (e.g., using PyBE) may offer an alternative that keeps the user in the loop, without the need for run-time user security decisions. Future work is needed in incorporating IFC enforcement into such highly automated systems that handle user resources.

The automation and lack of UI also have an unintended security consequence. Most application behavior in such systems is seldom preceded by a trigger from the user via a user interface (UI). As a result, a simple lack of UI activity before an application's resource request is insufficient to mark the request as suspicious, since the lack of UI activity is expected behavior. In contrast, on smartphones, an application's activities can generally be correlated with the user's interaction with the UI, and the lack of UI interaction may be assumed as a sign of a suspicious activity. Future work should adapt existing security mechanisms and heuristics to identify user-intended behavior without relying on the presence of UI triggers.

### 8.3.2 Network Export and Declassification

In IoT systems, devices are often resource constrained. Therefore, a lot of the processing happens on dedicated hubs, or in the cloud, with the cloud offering an advantage in terms of scalability. Additionally, applications may also want to run proprietary algorithms with the user’s data in their own cloud infrastructures. As a result, export of sensitive data to the network is not an exception, but a functional requirement in cloud-based IoT systems. Data may not be completely declassified on network export, and may need protection even after it is exported to the cloud. Prior work in this area requires programs or systems to be
significantly redesigned for distributed IFC guarantees [Liu09a; Zel08]. Future work may focus on extending light-weight IFC guarantees to the cloud, without prohibitive overhead or conditions on the cloud infrastructure.

8.3.3 Integrity Requirements

The requirement of integrity for the user’s resources is critical in IoT systems, much more so than in smartphones or PCs. While smartphones control the digital realm, IoT and smart homes affect physical objects in the user’s environment. A fire caused by a faulty oven or a fridge being turned off leading to food poisoning or unlocking the door in presence of intruders are all possible failures that can occur due to insufficient enforcement of integrity guarantees.

Future work should investigate the attack surface of high-integrity objects, and specifically, ensure that a low integrity input (e.g., from an untrusted application) may never affect a high integrity user resource (e.g., a door lock). Transitive enforcement may be used to provide strong integrity guarantees. However, future work may be needed to evaluate the practicality of such enforcement in these new environments (e.g., authenticating devices and tracking control flow through devices), and the feasibility of its policy specification (i.e., marking all input sources with integrity levels).

8.4 Concluding Remarks

On modern commodity operating systems, applications are designed to work together on the user’s behalf. Users can chain together applications for realizing complex tasks. However, operating systems do not allow users to enforce additional constraints on these chains, which exposes risks to user data secrecy. Without the ability to enforce constraints on user-directed chains of applications, users may have to sacrifice security for the value derived from using applications, or vice versa.

The techniques described in this dissertation demonstrate that it is possible to allow the user to share secret data with chains of unmodified third-party applications, without the risk of unauthorized disclosure. If the lessons learned from this research are applied to existing operating systems, prior approaches that limit user experience by restricting
access or use of applications may not be necessary. More importantly, these benefits are not limited to traditional, well-known user data (e.g., location, IMEI). By allowing the policy to be specified through applications that create user data, this dissertation facilitates the protection of application-specific user data (e.g., notes, document scans).

It is worth noting that the practical and secure enforcement primitives described in this dissertation are enabled solely by the novel runtime abstractions in modern commodity operating systems (e.g., mediated inter-application communication). Our success helps recognize the necessary traits in an operating system for feasible enforcement of strong security guarantees.

Finally, obtaining security guarantees for user data may involve more than just practical policy and enforcement techniques. The feasibility of specifying Policies by Example motivates future work in this area, and triggers fundamental questions in user-specific policy specification.
BIBLIOGRAPHY


173


Lineberry, A. et al. “These arenâ€™t the permissions youâ€™re looking for”. DefCon 18 (2010).


APPENDICES
We now prove the safety of the join operation in the Aquifer policy logic. Before proving the join operation ensures policy restriction, we must define a restriction relation. We do this in two parts. First, we define an effective restriction relation that ensures the evaluated policy is more restrictive. Then, we define an owner restriction relation that ensures that all of an owner’s restrictions are maintained. This is important, because while $L_2$ may be effectively more restrictive than $L_1$, an individual owner’s restrictions may be changed at a later time by another owner such that $L_2$ is no longer more restrictive than $L_1$. With these two definitions, we can define an overall restriction relation that is needed to prove the safety of Aquifer.

**Definition 9** (Effective restriction relation $\sqsubseteq_e$). Let $L_1$ and $L_2$ be workflow labels with effective export lists, required lists, and workflow filters $E_{1e}$, $E_{2e}$, $R_{1e}$, $R_{2e}$, $F_{1e}$, and $F_{2e}$, respectively.
$L_2$ is effectively more restrictive than $L_1$, denoted $L_1 \subseteq_e L_2$, if and only if:

\begin{align*}
E_{1e} & \supseteq E_{2e} \\
R_{1e} & \subseteq R_{2e} \\
\text{actions}(F_{1e}) & \subseteq \text{actions}(F_{2e}) \\
\forall s \in \text{actions}(F_{1e}), \text{targets}(F_{1e}, s) & \supseteq \text{targets}(F_{2e}, s)
\end{align*}

Conceptually, Definition 9 ensures that (1) $L_2$ has less exporters than $L_1$, (2) $L_2$ has more required apps on the workflow than $L_1$, and (3) any workflow filters in $L_1$ are enforced by $L_2$ with targets that are more restrictive (less than) those in $L_1$.

**Definition 10** (Owner restriction relation $\subseteq_O$). Let $L_1$ and $L_2$ be workflow labels, $O$ be the owner for which the relation is evaluated, $F_1 = \text{filters}(L_1, O)$, and $F_2 = \text{filters}(L_2, O)$. $L_2$ is more restrictive than $L_1$ for owner $O$, denoted $L_1 \subseteq_O L_2$, if and only if:

\begin{align*}
\text{exports}(L_1, O) & \supseteq \text{exports}(L_2, O) \\
\text{requires}(L_1, O) & \subseteq \text{requires}(L_2, O) \\
\text{actions}(F_1) & \subseteq \text{actions}(F_2) \\
\forall s \in \text{actions}(F_1), \text{targets}(F_1, s) & \supseteq \text{targets}(F_2, s)
\end{align*}

Conceptually, Definition 10 ensures the same properties as Definition 9, but with respect to owner $O$.

**Definition 11** (Restriction relation $\subseteq$). Let $L_1$ and $L_2$ be workflow labels. $L_2$ is more restrictive than $L_1$, denoted $L_1 \subseteq L_2$, if and only if $L_1 \subseteq_e L_2$ and $\forall O \in \text{owners}(L_1), L_1 \subseteq_O L_2$.

We now prove the safety of the Aquifer policy language.

**Theorem 1.** The Aquifer policy language is safe.

**Proof.** We prove the safety of the Aquifer policy language by construction. Let $L_1$ and $L_2$ be workflow labels. Workflow policy propagation creates a new label $L_1 \sqcup L_2$. We must show that $L_1 \subseteq L_1 \sqcup L_2$ and $L_2 \subseteq L_1 \sqcup L_2$. 
Based on Definition 11, \( L_1 \subseteq L_1 \cup L_2 \) iff (a) for all \( O \in owners(L_1) \), \( L_1 \subseteq O \) \( L_1 \cup L_2 \) and (b) \( L_1 \subseteq L_1 \cup L_2 \).

Condition (a) is satisfied by Definition 10 using Definition 8. For all owners \( O \in owners(L_1) \), let \( F_1 = filters(L_1, O) \) and \( F_2 = filters(L_2, O) \), then

\[
exports(L_1, O) \supseteq exports(L_1, O) \cap exports(L_2, O)
\]

\[
requires(L_1, O) \subseteq requires(L_1, O) \cup requires(L_2, O)
\]

\[
actions(F_1) \subseteq actions(F_1) \cup actions(F_2)
\]

\( \forall s \in actions(F_1), targets(F_1, s) \)

\[\supseteq targets(F_1, s) \cap targets(F_2, s)\]

Condition (b) is satisfied by Definition 9 using Definition 8 and applying Definitions 5-6 to determine the effective policy.

Export list: for \( L_1 \), \( E_{1e} = \bigcap exports(L_1, O) \) for all \( O \in owners(L_1) \). For \( L_1 \cup L_2 \), \( E_{12e} = \bigcap \{exports(L_1, O) \cap exports(L_2, O)\} \) for all \( O \in \{owners(L_1) \cup owners(L_2)\} \). To satisfy Definition 9, we must show \( E_{1e} \supseteq E_{12e} \). If an export list exists for an owner \( O_i \) in \( L_2 \) but not \( L_1 \), \( exports(L_1, O) \) will return the set of all applications (see Section 5.3) and the intermediate stage will be \( exports(L_2, O) \). However, if this contains an application that was not in \( E_{1e} \) it will be removed in the outer intersection. Therefore, \( E_{1e} \supseteq E_{12e} \).

Required list: for \( L_1 \), \( R_{1e} = \bigcup requires(L_1, O) \) for all \( O \in owners(L_1) \). For \( L_1 \cup L_2 \), \( R_{12e} = \bigcup \{requires(L_1, O) \cup requires(L_2, O)\} \) for all \( O \in \{owners(L_1) \cup owners(L_2)\} \). Clearly, \( R_{1e} \subseteq R_{12e} \), which satisfies Definition 9.

Workflow Filters: for \( L_1 \),

\[
F_{1e} = \{(s, T) \mid s \in \bigcup actions(F) \text{ and } T = \bigcap targets(F, s),
\forall F \in filters(L_1, O), \forall O \in owners(L_1)\}
\]
For $L_1 \sqcup L_2$,

$$F_{12e} = \{ (s, T) \mid s \in \bigcup \text{actions}(F_1) \cup \text{actions}(F_2) \}
\text{and } T = \bigcap \{ \text{targets}(F_1, s) \cap \text{targets}(F_2, s) \},$$
$$\forall F_1 \in \text{filters}(L_1, O), \forall F_2 \in \text{filters}(L_2, O),$$
$$\forall O \in (\text{owners}(L_1) \cup \text{owners}(L_2))$$

Definition 9 first requires showing that $\text{actions}(F_{1e}) \subseteq \text{actions}(F_{12e})$. This is true, because $F_{12e}$ contains all of the action strings in the filters for both $L_1$ and $L_2$. Second, we must show that $\forall s \in \text{actions}(F_{1e}), \text{targets}(F_{1e}) \subseteq \text{targets}(F_{12e})$. This is ensured by the intersection of targets when generating $F_{12e}$. This completes the conditions needed to satisfy Definition 9, as well as Definition 11 for $L_1 \subseteq L_1 \sqcup L_2$. The proof that $L_2 \subseteq L_1 \sqcup L_2$ follows similarly and is not shown for brevity. □
In this section, we describe Weir’s changes to the Activity Manager service’s component and process assignment logic. Figure B.1 shows the workflow inside the Activity Manager when a component C is called. The shaded blocks form Weir’s label checks and polyinstantiation logic. Note that the figure portrays the high level steps followed by the Activity Manager, common to all components. When a call arrives, Weir first gets the label for the caller’s process from the kernel and stores it in callerLabel. The Activity Manager then resolves the target component C using the information in the call. Note that at this point the Activity Manager only knows the name and type of the target component (e.g., the content provider C). The Activity Manager then checks if there is a runtime instance of C in its records, i.e., if C is running. If a runtime instance exists, Weir checks if the instance is running in a process with a matching (i.e., equivalent) label. If it is, then the call is delivered to the
running instance. Otherwise, Weir forces the Activity Manager to create another runtime instance, for this new callerLabel, i.e., polynomials the component.

Without Weir, the Activity Manager would always deliver the call to the existing instance.¹ Weir modifies the Activity Manager's internal bookkeeping structures to be consistent with its polynomination; i.e., it enables the Activity Manager to manage multiple runtime records for the same component. For example, the ActivityManager uses a direct mapping between a service's name and its runtime instance, to store records of running services. Weir modifies this mapping to one between the name and a set of services, each in a different process and with a separate label.

At this stage, the system has a new component instance that needs to be executed in a process. The Activity Manager selects the process based on the processName extracted from the “android:process” manifest attribute. A runtime record of the resolved process P is then

---

¹Except in the case of standard and multi-process activity components.

Figure B.1 Flow of the Activity Manager starting a component. The areas modified or added by Weir are shaded.
sent to Weir for process matching (i.e., the \textit{Match\_Weir\_Proc (P, callerLabel)} subroutine). Weir first checks the label of the existing process \( P \), and if it matches, returns \( P \) itself. If not, Weir retrieves its internal list of processes associated with \( P \). This list constitutes the processes that were created in the past to be assigned instead of \( P \) for specific caller labels. Weir checks if the list contains a process with a label matching the current \textit{callerLabel}; this step ensures that components with the same \textit{processName} as well as \textit{callerLabel} are executed in the same process. If Weir fails to find a matching process in the list, it allocates a new process for the \textit{callerLabel}, and adds it to the list of existing processes mapped to the specific \textit{processName}. This process is then returned as \( P \) to the Activity Manager. The Activity Manager then starts \( P \), if it is not already started, and Weir sets its label in the kernel. Note that if the process is already started (i.e., the original \( P \) was matching, or a matching process was found in Weir’s \textit{pList (P,processName)}, the Activity Manager does not restart it. Finally, the component instance is executed in the assigned process, and the call is delivered to it.

\textbf{B.2 Weir DIFC Policy}

Weir derives its policy structure from the Flume DIFC model [Kro07], which consists of \textit{tags} and \textit{labels}. A \textit{data owner (O)} application defines a security class for its sensitive data in the form of a secrecy tag \( t \). A set of such tags forms a secrecy label \( S \), i.e., \( S = \{t\} | t \in T \), where \( T \) is the universal set of all tags.

Weir enforces the classical IFC secrecy guarantee, i.e., “no read up, no write down” [BL73]. Information can flow from one label to another only if the latter dominates, i.e., is a superset of the former. For instance, data can flow from a process \( P \) to a process \( Q \) if and only if \( S_P \subseteq S_Q \). Weir applies this strict DIFC check to direct Binder communication and file accesses.

Like Flume, Weir also allows explicit label changes. A data owner controls the use of its tags in label changes through assignment of capabilities to other security principals. Each tag \( t \) is associated with two capabilities, \( t^+ \) and \( t^- \). The \( t^+ \) capability allows a process to add a tag to its label, enabling it to receive data protected with \( t \). The \( t^- \) capability allows removal of the tag from a label, declassifying the data associated with \( t \). Data owners can
delegate their capabilities to specific applications, or all other applications (i.e., the global capability set $G$). At any point of time, a process $P$ has an effective capability set composed of the capabilities delegated to its application ($C_P$), and all the capabilities in $G$. $P$ can change its label $S_P$ to $S_P^+$ by adding a tag $t$ if and only if $t^+ \in C_P \cup G$. Similarly, $P$ can change its label $S_P$ to $S_P^-$ by removing a tag $t$ if and only if $t^- \in C_P \cup G$.

Weir extends Flume’s syntax to support domain declassification, described in Section 6.5.4. The data owner specifies a set of trusted domains for each tag $t$ it creates; which is stored as the domain declassification capability $t^D$. The $t^D$ capability is not assigned to any security principal, but only used at the time of network export.

The network is untrusted, and hence has an empty label, i.e. $S_N = \emptyset$. Thus, a process $P$ must have an empty label (i.e., $S_P = \emptyset$), or the capabilities to hypothetically change its label to $S_P = \emptyset$. That is, $P$ can connect to the network if $\forall t \in S_P, t^- \in C_P \cup G$. For tags that may not be declassified in this manner, Weir uses its domain declassification. That is, when $P$ tries to connect to a domain $d$, Weir implicitly declassifies the tags whose domain declassification capability contains $d$. Like capability assignments, the data owner can also modify the set $t^D$ for every tag $t$ it creates.

### B.3 Application List for Compatibility Evaluation

We have listed the package names of the apps used for our compatibility test in Table B.1. The apps can be found on Google Play by replacing the `packageName` in the following URL with the actual package name: `https://play.google.com/store/apps/details?id=packageName`. 

193
Table B.1 Package names of singleTop, singleInstance and singleTask applications analyzed for compatibility.

<table>
<thead>
<tr>
<th>Single Top</th>
<th>Single Task</th>
<th>Single Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.netspend.product.android</td>
<td>com.fsp.android.friendlocator</td>
<td>com.ijinshan.kbatterydoctor_en</td>
</tr>
<tr>
<td>com.gpshopper.adidas</td>
<td>com.real.RealPlayerCloud</td>
<td>com.jk.dailytext</td>
</tr>
<tr>
<td>com.connectivityapps.hotmail</td>
<td>larry.zou.colorfullife</td>
<td>com.americanwell.android.member.amwell</td>
</tr>
<tr>
<td>com.ebay.mobile</td>
<td>com.pixlr.express</td>
<td>com.northpark.drinkwater</td>
</tr>
<tr>
<td>com.linkedin.android</td>
<td>org.lds.ldssa</td>
<td>com.intuit.quickbooks</td>
</tr>
<tr>
<td>com.handmark.sportcaster</td>
<td>com.escapistgames.starchart</td>
<td>com.northpark.pushups</td>
</tr>
<tr>
<td>com.cleanmaster.security</td>
<td>com.orbitz</td>
<td>com.camerasideas.instashot</td>
</tr>
<tr>
<td>com.yahoo.mobile.client.android.yahoo</td>
<td>com.pandora.android</td>
<td>com.linecorp.b612.android</td>
</tr>
<tr>
<td>com.babycenter.pregnancytracker</td>
<td>com.WaterfallLiveWallpaperHDHQ</td>
<td>com.fp.cheapoair</td>
</tr>
</tbody>
</table>
The appendix includes the scripts/user-instructions we used for the semi-structured interviews for data collection and our experiments. Additionally, all the tags used in our study can be found in Appendix C.1.

C.1  Tags used in the User Study

Figure C.1 shows the tags used in this study. We provided 9 tags, while the rest were created by users.

C.2  Collecting Examples

- In this task, you will provide context-policy examples.
You will be given a list of predefined context tags. You can use 0 or more of these tags, and also create your own tags.

You may combine tags to describe the context of a scenario. You will then be required to indicate a policy decision (i.e., 0 for deny and one for allow) for the policies provided.

Each line on the example sheet has space for the context (i.e., combination of tags), and a column for each policy.

### C.3 Collecting Weights

This phase consisted of two tasks, grouping tags and ordering groups. We provide the instructions given to users as follows:

#### C.3.1 Grouping tags

We provide an example of customized tag-group memberships in Figure C.2.
In this task, you will receive a spreadsheet containing groups, and information tags included in those groups. Note that the tags may include not only the tags you defined in the initial interview, but also tags defined by other users.

We have grouped tags that seem to be dealing with data of similar secrecy value. For instance, all the work-related tags such as “Work”, “WorkTravel” are in the group “Work”.

The task is to verify group memberships, such that every tag belongs to the correct group as per your understanding.

You can move tags around, i.e., remove them from one group, and add to another, but you cannot add, remove, or rename the groups themselves.

The spreadsheet will also include a comments column, if you want to make a comment about a specific group, although comments are not required.

Finally, the spreadsheet will include descriptions for some group names for reference.

### C.3.2 Ordering Groups

In this task, you will be given a set of partial order relations among the groups described in the previous experiment. This set of relations is only a baseline.
• A relation between groups A and B, such that A;B, means that B is more “important” than A. Since our policies are information secrecy-related, “more important” may be understood as “more sensitive”.

• Your task is to modify (i.e., add or remove) the given list of relations, i.e., to customize the orders according to the your data secrecy/privacy preferences.

• You will also be allowed to use your initial group assignment for reference.

• It is possible that the ordering of groups may be different for different policies. Therefore, you will be able to use different sets of relations for different policies (total 7 policies). Please indicate if you want to use the same order for all policies, or if you would prefer to use different orders. This choice can be made or modified at any point of time throughout this task.

• In the end, I will confirm each order, for each policy (if the user chooses to have different orders for different policies). For instance, if the user enters A;B, I will confirm “is B more important than A?”.

• Finally, you have the option of continuing to the next experiment after a break of 5-10 minutes, or calling it a day.

C.4 Review of Examples

The review phase consisted of two tasks, namely a manual review, and a semi-automated review using active learning. This section provides the scripts for both tasks.

C.4.1 Manual Review

• In this task, you will receive a set of spreadsheets (one per policy) containing your examples (i.e., the context label + policy decision) for that policy.

• This is an opportunity for you to review your examples, and modify the policy decision if necessary. The context labels cannot be modified.
• For each change you make, you will then indicate the cause of the change in the respective column of one of the following hints:

  – “I have changed my mind”: i.e., my policy preferences have changed.
  – “It seems I made an error before”
  – “I don't understand this policy example”: This could happen if you do not remember why you specified the policy, or are having trouble expressing it with tags you provided/used.

• Finally, for each change you make, please provide justification in the last column. This column may also be used for reasons other than the said hints.

• The investigator will go through the changes, and may ask you to provide any missing justifications or causes.

C.4.2 Semi-automatic Review

• In this task, our algorithm will suggest policy decisions for existing context labels that you have previously provided.

• You must either agree (y) or disagree (n) to the decision. You can also skip by entering n twice.

• For every decision that you disagree to, please provide a short justification.

• For example, the algorithm may suggest “Denial of export to the WorkCloud when the data object with the context created at Home, Photo”. This suggestion will be presented as follows: Home+Photos, WorkCloud = DENY (y/n)?

• If you agree, the algorithm will make another suggestion, or stop.

• If you disagree, the algorithm will provide a text input for the justification.

• The task will consist of at most 15 questions.
C.5 Testing with Random Examples

In this section, we describe the script for the phase of testing with random examples. This phase was split into two tasks as well, i.e., the task of labeling samples, and of the post test review.

C.5.1 Labeling Test Samples

• In this task, you will receive a set of spreadsheets (one per policy) containing examples (i.e., the context).

• Your task is to label the policy decision (allow/deny/I don't know) for each example.

C.5.2 Post-test Review

• In this task, our algorithm will ask you to confirm policy decisions that you have previously provided.

• Please agree (y) or disagree (n) with your decision.

• For every decision that you agree to, please provide a short justification.