ABSTRACT

NADKARNI, ADWAIT PRAVIN. Workflow Based Information Flow Control (IFC) in Modern Operating Systems. (Under the direction of Dr. William Enck.)

The security architecture of consumer operating systems is currently undergoing a fundamental change. In platforms such as Android, iOS, and Windows 8, each application is a separate security principal that can own data.

While this distinction is a vast improvement over traditional user-focused security architectures, sharing data between applications results in an unexpected loss of control of that data, potentially exposing security and privacy sensitive information. This paper presents the Aquifer framework to address this data intermediary problem in modern operating systems.

Aquifer builds from the notion that when apps share data, it is as part of a mashup to perform some task for the user. As such, Aquifer enforces host export restriction policies on data associated with the user interface workflow. These restrictions are specified by the applications on the UI workflow using available runtime context, as opposed to relying on users for manual specification. In doing so, Aquifer allows applications to retain control of data even after it is shared, thereby addressing a growing concern in modern operating systems.
© Copyright 2012 by Adwait Pravin Nadkarni

All Rights Reserved
Workflow Based Information Flow Control (IFC) in Modern Operating Systems

by
Adwait Pravin Nadkarni

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Douglas Reeves
Dr. Xuxian Jiang

Dr. William Enck
Chair of Advisory Committee
DEDICATION

To my parents.
BIOGRAPHY

The author has completed his Bachelors Degree in Computer Engineering from the University of Mumbai, India, and joined the NC State University as a Masters student (MR-CS) in Fall 2011.
ACKNOWLEDGEMENTS

I would like to thank Dr. William Enck for being a great advisor and mentor. This thesis would not have been possible without his help and guidance. I also thank Dr. Xuxian Jiang and Dr. Douglas Reeves for taking the time to be on my committee, and for being supportive of my research. I thank Anson Ho, Vasant Tendulkar, Ashwin Shashidharan and Sigmund Gorski for their valuable inputs on this work.
# TABLE OF CONTENTS

List of Figures .......................................................... vii

Chapter 1 Introduction .................................................. 1
  1.1 Thesis Statement .................................................. 4
  1.2 Contributions ..................................................... 4
  1.3 Organization ....................................................... 5

Chapter 2 Background: Android ......................................... 6
  2.1 Application, and Components .................................... 6
  2.2 The Activity Lifecycle ............................................ 7
  2.3 Intent Resolution .................................................. 8
  2.4 Android’s Binder IPC .............................................. 8

Chapter 3 Motivation ..................................................... 10
  3.1 Contract Document Use Case ..................................... 10
  3.2 Policy Requirements ............................................... 12
  3.3 Example Policies .................................................. 12
  3.4 Threat Model and Assumptions ................................... 12

Chapter 4 Overview of Approach ....................................... 15
  4.1 Challenges ......................................................... 16

Chapter 5 Policy Design ................................................ 20
  5.1 Restrictions Types ................................................ 20
    5.1.1 No Restrictions .............................................. 21
    5.1.2 Export ....................................................... 21
    5.1.3 Filters ...................................................... 22
    5.1.4 Required .................................................... 23
  5.2 The Aquifer Policy ............................................... 24

Chapter 6 Aquifer Design ............................................... 28
  6.1 Filtering UI Workflows .......................................... 28
  6.2 Maintaining the Workflow State ................................ 30
  6.3 Network Access Control ......................................... 31
    6.3.1 Assigning the network access privilege .................. 32
    6.3.2 Enforcing the network access privilege .................. 33
  6.4 Policy propagation to storage .................................. 33
    6.4.1 Protecting Files: ........................................... 34
    6.4.2 Propagating policy through daemons: ..................... 34
Chapter 7 Implementation ........................................ 39
  7.1 The Aquifer Service ............................................ 39
    7.1.1 Maintaining information on workflows ................. 40
    7.1.2 Workflow Filter using Aquifer Service ................ 41
    7.1.3 Adding and removing components from a workflow .... 42
    7.1.4 Setting the Network Access Policy ..................... 43
    7.1.5 Maintaining labels on Files ............................ 43
    7.1.6 Adding Application Policies ............................ 44
  7.2 Aquifer Device and LSM ...................................... 44
  7.3 The Aquifer API ............................................... 45

Chapter 8 Evaluation .............................................. 46
  8.1 Performance .................................................. 46
    8.1.1 Security Evaluation ...................................... 47
    8.1.2 Case Study: K-9 Mail .................................... 51

Chapter 9 Related Work ........................................... 54

Chapter 10 Discussion ............................................. 56

Chapter 11 Conclusion ............................................ 58

References .......................................................... 59
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The lifecycle of an Android Activity [2]</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Document signing use case. A confidential contract received via Email before being converted to PDF and embedded with a written signature.</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>A conceptual overview of the Aquifer Architecture</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>Block Diagram of the Aquifer System. Shows basic interaction between Aquifer and a UI Workflow</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Workflow filter in action: Policy enforcement in visual Workflows</td>
<td>37</td>
</tr>
<tr>
<td>6.3</td>
<td>Setting the network access privilege for components using the Aquifer Network Access Controller.</td>
<td>38</td>
</tr>
<tr>
<td>7.1</td>
<td>Software Stack of the Android Operating System patched with Aquifer.</td>
<td>40</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Operating system security architectures are currently undergoing a fundamental change. Traditional commodity OSes run all of a user’s applications with the same privilege, allowing access to all of the user’s files, and the files of all of the user’s applications. Modern operating systems [33, 40], such as Android, iOS, and Windows 8, take the suggestion of decades of security research [41, 23, 37, 16] and run each application as a unique security principal.

If applications operated in complete isolation, protection in this new OS architecture would be straightforward (i.e., sandboxing [19, 18, 32]). However, this is not the case. Consider the smartphone platforms that are leading the renaissance to application-based security principals (e.g., Android). Applications, known as “apps,” are frequently purpose or service specific (e.g., a barcode scanner or a social networking app). While apps frequently work in isolation, they are increasingly chained together, providing a workflow that encompasses a mashup of user-specified services. For example, a shopping app might: 1) invoke a barcode scanner app that uses the camera to read the UPC from an item in a retail store, 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).

Data sharing between user apps represents a primary security challenge for modern OSes. When one app shares data with another app, or writes data to storage accessible by other apps, it loses control of that data. This *data intermediary problem* has always been present in operating systems, but addressing it made little sense when all user apps ran with the same privilege. Now that apps are security principals, the data intermediary problem must be addressed. This is a significant challenge because acceptable chains of interactions are based on user selection and hence are difficult to predict. The current security mechanism, *permissions*, is insufficient. Permissions are boolean access control policies specified on interfaces. They do not account for multiple apps accessing data through a (potentially) user-defined workflow. This lack of transitive semantics is the root cause behind attacks on Android’s permission system [10, 17, 12, 20].

Previous attempts to add transitive security semantics to Android have focused on integrity [17, 12, 6]. Systems considering secrecy (e.g., TaintDroid [15] and AppFence [22]) have focused on user privacy. Our focus is on *application-specific user data*. Apps originate many new types of data that require protection (e.g., office documents received via email or cloud storage, voice and written notes from meetings, address book entries, social networking events, health records, inventory tracking information, etc.). Therefore, we seek to create mechanisms and policy that allow participation by both users and app developers.

A secondary goal of this work is to minimize the need for asking users security questions. Recent work has shown how to use trusted UI widgets to infer user intentions when accessing sensors (e.g., camera, microphone) [33]. This approach builds upon prior work that captures user intentions using trusted dialog boxes for choosing files [13]. We see
great opportunity in new platforms such as Android to extend these concepts to support a decentralized notion of trust and policy specification [29, 39, 44, 27]. In Android, the user’s UI workflow traverses multiple applications to access data, and the owners of specific data types are invoked to access that data (e.g., the photo gallery app defines the chooser for attaching pictures to an Email). Apps that own data can infer user intentions from the UI workflow and attach security policies to data that include app-specific context unavailable to centrally defined trusted UIs.

This thesis proposes Aquifer, a novel security framework that provides data secrecy policies for application-specific user data in modern operating systems such as Android. Aquifer defines security policy with respect to the users UI workflow, thereby tracking user data as it traverses through user-directed interactions between apps. Apps contributing sensitive data can include security policy that restricts the further use of that data. In doing so, we begin to directly address the data intermediary problem in modern operating systems.

We have implemented Aquifer for the Android platform. We specifically leverage the artifact of Android that app functionality is divided into many components (e.g., activity, service, broadcast receiver, and content provider). This inherent division between functionality allows simple privilege separation: components can frequently be run as separate processes. This enables sub-application protection domains for defining security policy, striking a balance between application-level tracking and variable or byte-level tracking (e.g., as in TaintDroid [15]). We have provided two sample policies for applications and evaluate three apps that simulate real world applications to demonstrate the practicality of Aquifer.
1.1 Thesis Statement

“Tracking information via User Interface (UI) Workflows provides developers with a practical means of maintaining control over application specific data.”

1.2 Contributions

This thesis presents Aquifer, an novel security architecture that provides transitive data secrecy policies for application-specific user data in modern OSes. Aquifer enables data secrecy policies for app-specific user data defined with respect to a UI workflow. Prior security enhancements to Android have focused on system defined resources and data and provide no protection for app-specific user data. Aquifer also allows apps to define the security policy that protects the data they own.

A direct consequence of the enforcement of such a policy is a policy violation. Most research in this area handles policy violations by presenting the user with a prompt that details the cause of the violation, and its consequences, leading to user confusion. Some approaches silently drop such policy violations, leading to application crashes due to unhandled exceptions. Both these approaches lead to a bad user experience, and decrease the usability of the security solution. Aquifer, on the other hand, presents an approach to prevent access control violations from happening in the first place, using a technique we call ‘Workflow Filtering’.

We provide a proof-of-concept implementation of Aquifer for Android. We show that Android’s architecture inherently allows app partitioning by components. We then use component-level tracking to balance performance and accuracy. We provide two sample data secrecy policies and evaluate Aquifer using three applications. We develop inter-
mediaries that export data, and demonstrate how the *Aquifer Architecture* provides the level of security expressed by applications through the security policy.

### 1.3 Organization

The remainder of this thesis proceeds as follows. Chapter 3 provides a use case for Aquifer and defines our threat model, and also presents some example policies. Chapter 4 overviews our approach. Chapter 5 explains the Aquifer policy, while Chapter 6 describes the Aquifer design. Chapter 8 describes the evaluation for Aquifer through experiments.
Chapter 2

Background: Android

This work has been implemented in Android, and hence most of the major aspects have been described with respect to Android. These concepts are transferable to most modern operating systems.

2.1 Application, and Components

The Android OS is a multi-user Linux operating system, where each application is a separate user. Each application can be composed of four component types: activities, services, content providers, and broadcast receivers. Of these, activities are the only components which present a user interface and directly interact with the user, while the rest are background components. Since this work is about securing User Interface(UI) workflows, our main focus will be on activities throughout this discussion.
2.2 The Activity Lifecycle

An activity transitions between 4 states, namely STARTED, PAUSED, STOPPED and DESTROYED. A paused activity can be resumed, and a stopped activity can be restarted; but the DESTROYED state is a point of no return; i.e., it is not possible to execute any code in the activity after it attains the DESTROYED state, as shown in Figure 2.1.

![Figure 2.1: The lifecycle of an Android Activity](image)

Figure 2.1: The lifecycle of an Android Activity [2]
2.3 Intent Resolution

Android supports asynchronous inter-component communication through Intents. An intent is a message from one component, which triggers some execution in(or even the initiation of) another component. There are two kinds of intents, explicit and implicit. Explicit intents are generally used for intra application communication, and are targeted at a particular component specified by its full class name.

Implicit intents, on the other hand, specify the action or service required by the calling component, using “action strings”. The android operating system then forms a list of suitable components that have previously registered to fulfill that particular request or perform the particular action specified in the intent. Apart from the action, there are a number of other elements such as the category, type of data, etc involved in the resolution of an intent, but we do not discuss them to keep the discussion simple. Activities are started using the startActivity or startActivityForResult method calls. In case there are more activities that perform the action specified in the intent, Android presents the user with the list of activities to choose from. Upon selecting an option, the user is presented with the chosen activity. Intent resolution is performed using a class called IntentResolver by Android’s ActivityManagerService.

2.4 Android’s Binder IPC

Android’s runtime environment is based on Binder. Binder provides a system-level component abstraction on top of the standard process abstraction provided by traditional operating systems. In binder, applications are built as collections of component threads that do not necessarily reside in the same process.
Binder transactions are synchronous IPC addressed to a *binder object reference* to an interface defined by the target process. The source process specifies the target binder reference, a predefined transaction code, and a data parcel. This information is passed to the binder kernel module using an ioctl, which blocks for a reply. The binder kernel module uses internal state to dereference the target process from the binder reference. Upon receiving the transaction, the target process uses the supplied transaction code to pass the data parcel to the correct interface method. The interface method populates (if necessary) a reply parcel and passes it to binder. The binder kernel module then delivers reply to the source process’s binder thread, unblocking it. In order to share files between processes, Binder also allows transfer of “file descriptors” through it.
Chapter 3

Motivation

Aquifer is motivated by the data intermediary problem in modern operating systems where users perform tasks by combining the functionality of multiple apps. We begin this section by describing an example use case requiring transitive information protection. We then describe the policy requirements of applications that take part in this use case, and some example policies. We conclude this section by defining our threat model and assumptions for the Aquifer design and implementation.

3.1 Contract Document Use Case

Modern operating systems enable a mashup of apps that implement a larger user task. The resulting UI workflow can be a simple linear progression, or a more complex arrangement where subtasks are performed to obtain data before returning to a previous stage to continue the primary task. For simplicity, this use case presents the former, depicted in Figure 3.1.

Alice receives a confidential contract in her business Email. She needs to sign and
Figure 3.1: Document signing use case. A confidential contract received via Email before being converted to PDF and embedded with a written signature.

return the contract. Alice is traveling and does not have access to a printer or a scanner; therefore, she uses the DocuSign app on her smartphone to digitally attach a written copy of her signature. The workflow begins by Alice accessing the message containing contract.doc in the Email app. Alice selects to read 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 first shares contract.doc with the WordToPDF app to create contract.pdf, and then shares contract.pdf with DocuSign. She then uses DocuSign to embed a copy of her written signature, creating signed.pdf. This file is then shared with the Email app to send the signed contract.

This use case includes two assets: the contract document, and the copy of Alice’s written signature. The contract document is owned by the Email app, and the written signature is owned by DocuSign. While the Email app can identify the contract document as confidential (e.g., via Email message headers), if the above task is to be performed, it must share the document with other applications. Therefore, the Email app might wish to specify policy which ensures that other applications also treat the document as confidential. On the other hand, the DocuSign app might want to prevent reuse of the signature, as well as prevent some intermediary application from storing the signed document and later exporting it out of the device.

11
3.2 Policy Requirements

As seen in the use case above, an application which owns data wants to share it with other applications, but also wants to prevent its misuse by the receiving application. An Email app like Gmail would not want to prevent a confidential attachment from being exported to any website other than itself. On the other hand, a document signing app like DocuSign may want its signed document to only be used in its current workflow, and not be cached and exported later by another application.

3.3 Example Policies

Considering the security requirements of the applications involved in the use case, we present some security policies:

Policy (a): Allow other apps to view and edit contract.doc, but only the Email app may send contract.doc and derivative files off the device.

Policy (b): Documents signed by DocuSign can only be sent off the device if accessed in a workflow that contains DocuSign. This policy prevents a cached copy of signed.pdf from being later discovered and exfiltrated, or the signature from being reused.

3.4 Threat Model and Assumptions

quifer is designed to protect application-specific user data. Apps originate many new types of data that require protection (e.g., office documents received via email or cloud storage, voice and written notes from meetings, etc.). Since this data is introduced by third-party applications, it is unknown to the platform security policy and therefore cannot be protected by it. Furthermore, it is unreasonable to expect the user to define
complete security policies for data, or to always use the data in a safe way. Therefore, application developers must participate in the definition of security policy for their data. Hence, the applications (including third-party apps) trusted for a data item is defined by its security policy.

In this paper, we specifically focus on large data objects that are stored in files. We make this simplifying assumption to reduce the complexity of the protection system so that we may concentrate on the policy semantics for leveraging UI workflows for policy specification. However, we believe that the conceptual foundations can be extended to protect finer granularity data, which will be investigated in future work. Chapter 6 discusses this assumption further.

Both the user and the application defining an Aquifer policy are stakeholders for the corresponding data. While Aquifer could be used to define digital rights management (DRM) policy, DRM is not an explicit capability of Aquifer. We do not attempt to defend against a physical user determined to extract data. Aquifer allows multiple applications to add policy restrictions to data. We ensure that each application’s policy is enforced by determining the least restrictive policy that upper bounds the policies. Note that in doing so, we prioritize secrecy over availability. We explicitly do not address malicious applications that define security policies to make data inaccessible.

Aquifer protects against undesired data exposure by applications that receive data as a result of user choices (i.e., sharing data within a mashup of apps). The data exposure may result from a malicious recipient application exfiltrating the data, or inadvertent exposure when a user shares data with an app without understanding the consequences (e.g., a social networking app). As a consequence, Aquifer can prevent social engineering attacks on data sharing between applications that result in data exfiltration (e.g., confusing the user via the share chooser).
Aquifer minimizes user specification of security policy by inferring user intention from the UI workflow. While this reduces the burden on the user, it does not entirely eliminate it. Sometimes the application defining Aquifer policy must distinguish between confidential and public data. In these cases, we assume the existence of preliminary labeling. This ranges by application. For example, the Email app in our use case could determine secrecy requirements from an Email header set by the sender. Other applications such as note apps (e.g., Evernote) already have semantic tags on data (e.g., business, personal) that can be leveraged. However, in other cases (e.g., DocuSign), the policy specification is inherent to the functionality of the app. In this paper, we assume the app defining Aquifer policy has sufficient information to define the policy and leave the usability of labeling data in different scenarios to future work.

Finally, our trusted computing base (TCB) includes the operating system and its services. The OS is assumed to be vulnerability free, and root privilege escalation not possible. While there has been increasing concern of root privilege escalation in the Android platform on which our prototype system is built, ensuring the integrity of the OS is a common assumption for systems security proposals and is outside the scope of this work.
Chapter 4

Overview of Approach

Application components can be classified into two types based on their interaction with the user: the User Interface components (UI components) and background components. The user directly interacts with the UI components, while only interacting with the background components through the UI components.

Essentially, a UI workflow is the series of screens the user navigates through to perform a task. We assume that each of these screens corresponds to a separate piece of code, or workflow component (e.g., activities in Android applications), which can span multiple applications. UI workflows are transient, existing only as long as the task needs to be completed; but components of the workflow may also interact with background components (e.g., deamons). Workflows can also interact with the network, through workflow components which display web data to the user/export user data to the cloud. A workflow may not always be linear, and may contain branches. Additionally, depending on the windowing environment, there may be multiple simultaneous workflows on the host.

Because a UI workflow is guided by the user’s actions, it represents how the user is using the information, and provides applications with knowledge of the context in which
information is being used. Aided by this context knowledge, applications can influence the UI workflow policy to control the flow of data they inject in the workflow. The rest of this section states the general challenges faced by workflow based information flow control systems like Aquifer, and describes how the Aquifer Architecture overcomes each of these challenges.

4.1 Challenges

Enforcing security policy on UI workflows has the following challenges:

- **Expressing appropriate policy semantics.** We need a policy design that allows components to express their constraints on the workflow with sufficient accuracy, and which is also simple to understand.

- **Multiple policy owners.** Security sensitive information can be introduced into the UI workflow at any point. When this happens depends on application logic. Each application participating in a UI workflow should be able to contribute to the security policy of that workflow.

- **Minimizing policy violations.** Policy violations confuse users by either prompting the user to make security decisions, or breaking application functionality (e.g., a SecurityException causes an Application Not Responding (ANR) prompt and crashes an app in Android).

- **Propagating policies within the UI workflow.** Every component participating in the workflow should be subject to the restrictions placed on the workflow via the workflow policy.
We present the Aquifer Architecture to tackle these challenges. Aquifer consists of a UI-workflow abstraction for tracking associated program execution across applications. This requires creation of a workflow abstraction in the operating system, and adding components to it based on the user’s interaction with the UI. Additionally, the state of
the workflow needs to be consistent with the state of the UI. *Aquifer* also associates a security policy with the workflow, and enforces it on all the participants of the workflow.

Our architecture enables each participant in the workflow to add to the workflow’s policy, and to manipulate its part of the policy. This also requires maintaining distinct policies for each participating app, to isolate policy modifications based on ownership. Access control policy violations are reduced by modifying the navigation options presented to the user.

In the Figure 4.1, there are two workflows. The app X in the workflow at the top sets a policy on its data. This policy is then applied to the entire workflow it is a part of. Observe that the policy also propagates to another UI workflow component, through a background component, and then applied to the bottom workflow. Thus, we can say that *Aquifer* also tracks UI data flow through system daemons.

The transient nature of our UI workflow security policy is very similar to capabilities [42, 25, 5, 24, 36]. As shown in the Figure 4.1, components in the UI workflow can add policy to the workflow, and can potentially affect applications that later appear in the workflow. If there was a branch in the workflow, and if a component in the workflow received data from the branch, later components would still be affected by the data from the branch and hence its policy; which is similar to History Based Access Control (HBAC) [14, 1]. However, our policy structure is more expressive than capabilities and our policy enforcement provides transitivity, unlike traditional capability systems.

Since we deal with application-specific data, our system enforces policies contributed by many applications. This was motivated by decentralized information flow control (DIFC) systems [39, 44, 45, 27, 26, 38, 29], since applications as principals design their own security policies. However, existing DIFC systems are difficult to work with, as they have a complicated policy arithmetic and enforcement logic. Further, they do not necessar-
ily correspond to the user’s UI workflow, in which case they do not utilize the context knowledge that comes from the UI.
Chapter 5

Policy Design

A major component of any information flow control system[29, 44, 45, 27, 39] is its policy. An expressive policy design is essential for applications to be able to convey most of their security requirements. At the same time, the policy design should not be too complicated, such that it is unusable.

We attempt to strike a balance with our Aquifer Policy, which helps applications express their major security requirements, but is simple enough to understand and configure. We start this section by describing the types of restrictions applications might want to impose on the use of their data. Then, we describe the Aquifer policy design, and the policy language associated with it.

5.1 Restrictions Types

The document signing example in Section 3 motivated two types of security requirements: secrecy and regulate. In this section, we describe these security requirements with respect to their desired policy semantics. We also discuss Aquifer's default behavior of placing
no restrictions.

5.1.1 No Restrictions

The default behavior of Aquifer is to place no restrictions on the UI workflow. While default allow policies are generally frowned upon for security frameworks, Aquifer does not seek to control all data in the OS, just specific types of high-value application-specific data. For example, applications frequently share public URLs with other applications (e.g., sharing a link from a Web browser to Facebook or Email). Such scenarios do not require security policies. Additionally, for UI workflows that have an existing policy (e.g., Figure 3.1), applications contributing to the workflow frequently have no need to contribute additional restrictions, even if they add data (e.g., if DocuView allowed Alice to modify the contract in Figure 3.1). By using a default allow policy, Aquifer is more compatible with legacy applications that do not have security requirements.

5.1.2 Export

In OS access control literature, secrecy policies are traditionally associated with the transitive extension of read access. That is, secrecy policies allow an app to read data, but once the data is read, the app is restricted in what it can do with the data. Secrecy policies require data to be tracked at some granularity (e.g., byte-level or process-level). For process-level tracking, the reading process becomes “tainted” with the secrecy label on the data. The most common type of secrecy policy is an export restriction, i.e., once an app reads protected data, it cannot communicate via the network interface. Export restriction policies allow any functionality on the host, but prevent leakage to remote parties that are not mediated by the framework.
Aquifer is explicitly motivated by export restriction policies. A data owner application on the UI workflow specifies an export restriction by listing the applications that are allowed to send the data off the host, i.e., it specifies a whitelist. Therefore, as soon as an export restriction is placed on a UI workflow, Aquifer becomes default deny (instead of default allow) for that UI workflow.

Specifying the whitelist requires the app defining the restriction to be aware of other apps that need to export the data. We expect the document signing use case in Figure 3.1 to be a common scenario. Here, the Email app specified a whitelist of only itself. However, there may also be scenarios where a suite of applications is commonly used together, and the developer will know the whitelisted apps a priori. Note that such prerequisites are common for security policies (e.g., Saint [31]).

### 5.1.3 Filters

Related to export restrictions are UI workflow filters. Aquifer allows applications to define UI workflow filters specifically to enhance usability. One of the features of Android that enables a rich mashup experience is its *intent* message resolution. When starting the next screen (i.e., activity component) on the UI workflow, an app can specify an “action string” (e.g., `ACTION_SEND`, `ACTION_VIEW`, `ACTION_EDIT`) that helps 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 (either malicious or benign) 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 allows a data owner to specify a UI workflow filter that limits the potential targets of Android’s action strings. For example, the Email app in Figure 3.1 might specify that `ACTION_SEND` can only resolve to itself. Note that Aquifer includes UI workflow filters entirely for usability and does not seek to prevent manipulation by malicious applications (e.g., sending as a result of `ACTION_VIEW`).

5.1.4 Required

The current state of the workflow can provide applications with runtime context. For example, information about what applications have been a part of the workflow or are currently participating, might be utilized by apps to configure the necessary restrictions.

Applications can restrict the export of data based on the context of the workflow. Such Context based Export Control allows applications to restrict export of cached data. For example, in the Contract Document use case from Chapter 3, suppose the user opens the signed contract in an application M, from the DocuSign app. This app M could cache the document, and finish. Then it could start a new instance of itself which is no longer a part of the workflow, and hence no longer bounded by the restrictions placed on the workflow, and export the cached copy of the signed contract document. To prevent such a situation, the DocuSign app might want to restrict the export of its data to only a workflow which it is a part of, thus ensuring that cached copies are not misused. Enforcing such a restriction would require the runtime context of the workflow, specifically knowledge of the apps that have led the workflow to its current state.
5.2 The Aquifer Policy

The Aquifer policy logic incorporates the restriction types just described. Our logic is motivated by the logic described for the decentralized label model (DLM) [29]. We chose this logic from amongst the various DIFC logic proposals [29, 39, 44, 45, 27, 26, 38] 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 specifically to control data export, and does not enforce interaction between applications at runtime. Rather, UI workflow policies propagate in the style of taint tracking.

Aquifer UI workflow policy defines security principals as applications. Using UI screens as principals becomes cumbersome to specify and manage, and is often impractical since developers defining security policy do not necessarily know the UI screens in other applications. The UI workflow policy is a collection of owner policies, where each owner is an application. The owner policy contains an export list, a workflow filter, and a required list.

**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** (Workflow filter). A workflow filter $F = \{(s_1, T_1), \ldots, (s_n, T_n)\}$ is a set of tuples, 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 applications $N$, then the resulting allowed target applications is $N \cap T_i$.

To simplify the following discussion, we define functions for retrieving the the action string an set of targets from a workflow filter. For a filter $F$, $actions(F)$ 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).

**Definition 3** (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.

Having defined export lists, workflow filters, and require lists, we can now define a workflow label.

**Definition 4** (Workflow label). A workflow label $L$ is an expression

$$L = \{O_1 : (E_1, F_1, R_1); \ldots; O_n : (E_n, F_n, R_n)\},$$

where $O_i$ is an owner (application) and $E_i$, $F_i$, and $R_i$ are an export list, workflow filter, and required list, respectively.

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., $O_1$ cannot change $E_2$, $F_2$, or $R_2$).

We define functions for retrieving the parts of an owners policy from a label $L$. Here, we must be careful, since Aquifer assumes a default allow policy and only turns to default deny if a restriction is specified by an owner. Let the set of all applications be $\mathcal{A}$, and the set of all possible action strings be $\mathcal{S}$. 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 $\mathcal{A}$. 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 \( \{(s, A) | \forall s \in S\} \). 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, workflow filter, and required list 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 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, \( F_e = \{(s_i, T_i) | s_i \in \bigcup \text{actions}(F) \text{ and } T_i = \bigcap \text{targets}(F, s_i), \forall F \in \text{filters}(L, O), \forall O \in \text{owners}(L)\} \).

**Definition 7** (Effective required list). For a workflow label \( L \), the effective required list \( R_e = \bigcup \text{requires}(L, O), \forall O \in \text{owners}(L) \).

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 a least restrictive label that maintains all of the restrictions specified by \( L_1 \) and \( L_2 \) [29].

**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:
owners(L) = owners(L₁) \cup owners(L₂)
exports(L, O) = exports(L₁, O) \cap exports(L₂, O)
requires(L, O) = requires(L₁, O) \cup requires(L₂, O)
filters(L, O) = \{(sᵢ, Tᵢ) | sᵢ \in actions(F₁) \cup actions(F₂),
Tᵢ = targets(sᵢ, F₁) \cap targets(sᵢ, F₂),
where F₁ = filters(L₁, O),
F₂ = filters(L₂, O)\}

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 L₁ and L₂, 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.
Chapter 6

Aquifer Design

This chapter describes the core design concepts of our approach, and also the resolution of the design challenges presented in Chapter 4. Though the Aquifer system has been implemented for the Android OS, the core design concepts are easily transferable to other emerging operating systems which view applications as principals.

Our design can be abstracted into the four main tasks, i.e.: 1) Filtering UI Workflows, 2) Maintaining the Workflow State, 3) Network Access Control, 4) Policy propagation to storage.

6.1 Filtering UI Workflows

The Workflow label stores the constraints set on the entire workflow. Thus, in order to enforce these constraints, every interaction by every component that is a part of the workflow must be intercepted and checked against the Workflow Label L. A UI workflow progresses as a user requests the active component to perform a task. If the active component cannot perform that task, it asks the operating system to suggest an alternative
application(s) to the user that can perform the requested task. The user selects an application from the list presented by the OS, and the workflow includes the newly started application component. The workflow is constrained by policies set by applications whose components have participated in the workflow. These policies limit the privileges of components added to the workflow. Hence, there is a possibility that the selected component may try to perform the requested operation, but may not permitted by the policy on the workflow, Aquifer would intercept this attempt and throw a security exception. Such security exceptions result in application crashes and user confusion, and over all a very bad user experience.

On the other hand, if the operating system filters out applications which are not permitted to perform the requested action before presenting the list to the user, the user will only select apps that will not lead to a security exception. For example, consider the scenario (a) in Figure 6.2 where the Gmail app has denied the Facebook application the privilege of exporting its data off the device. The user tries to share(export) a pdf document attached to an email, due to which Android presents a list of applications that can service the user’s request. This list contains the Facebook app as it has registered for providing sharing services. If the user selects Facebook, a policy violation will occur as soon as Facebook tries to access the network, resulting in an application crash. Instead, in scenario (b) of the same Figure 6.2, Aquifer filters out applications which may conflict with the workflow label, according to the Workflow Filter(F) component of the effective policy on the label, and then presents the final list to the user. In the figure, the Gmail application’s policy states that only the Gmail application itself can receive the pdf document with an intention to share or send it. The Aquifer system filters the resolved list accordingly. This reduces the probability of a policy violation in the future. Since applications are data owning principals, we do not perform any kind of filtering on the
communication between the components of the same application, though they are also added to the workflow.

6.2 Maintaining the Workflow State

The workflow’s state consists of the information about the components participating in the workflow. As a user moves forward along a UI workflow, new application components are started with data from the workflow, and *Aquifer* ensures that these components are added to the workflow’s label. We maintain a separate list of components which are currently active in the workflow. The newly started component is added to this list. Thus, we ensure that every component which potentially receives data from the workflow is also subject to the restrictions placed on the workflow.

The workflow’s state also contains the list of policies that have been added to the workflow by *Owners* of the data flowing in the workflow, i.e. applications. Applications add policies to the workflow, when their instances are executing in the workflow. An application might want to add more restrictions, for which it again adds a new policy to the workflow, which replaces the original policy. An application might also want to declassify some of its data, and might add a less restrictive policy to the workflow. This new policy again replaces the old policy for the application, i.e. the *Owner*. Applications use the Aquifer API to build policies. We also export some methods of the *Aquifer Service*, which allow applications to upload their policy to the UI Workflow’s label. The only reason we can successfully allow applications to modify just their own policies in the Workflow’s security label is because we isolate policies within the workflow label based on the Owning principal. An application component can only modify the policy of its owning principal, i.e. application.
When the user exits or closes a component she was interacting with, the OS performs reclaims some resources allocated to this component. Similarly, in order to keep the state of the workflow consistent with the state of the system, and in order to apply only the minimum required restrictions on the workflow, we have to perform a set of tasks after a workflow component exits. First, we check if the component that is exiting returns any data to the component that started it. If it does not, then it means that the exiting component is not contributing any data to the workflow’s state after it exits. In this case, we also check if any other components from the same application, i.e. the owner, exist in the workflow. If there is no other component from the same application as of this exiting component, and if this component is not contributing data to the workflow in any way, we remove its application’s policy from the workflow’s label. This ensures that a policy will affect the workflow only if data associated with the policy is present in the workflow. Along with this, we then remove the exiting component from the list of currently active components in the workflow, and add it to the list of historical components. When the last component in the workflow exits, the workflow is destroyed.

6.3 Network Access Control

Most applications installed on smartphones have the privilege to access the network, and data can easily be exfiltrated to some remote server by any application that receives it. Hence, it is essential to offer applications a way of specifying a subset of applications that can export their data, the conditions that are to be met before the data can be exported. We provide this control via the Export list($E$) and the Required List($R$) of the Aquifer policy. Aquifer then enforces the policy set by the application through its Network Access Controller.
As described in the policy design, the Export list of a policy lists the applications that have the privilege of exporting the Owning principal’s data. The Required List on the other hand, restricts network access to all components present in the workflow until all the applications listed are a part of the workflow.

The Network Access Controller we define performs two main functions: 1. Assigning and Revoking the network access privilege for applications, based on policy changes in the workflow; and 2. Enforcing the network access checks to ensure that only privileged applications get network access.

6.3.1 Assigning the network access privilege

Workflow components may make modification to their application’s Owner policy in the workflow’s label, and may thus declassify some data for sending by adding some principals to the Export List, or deny the sending privilege to some applications by removing some principals from the Export List. An app may also add some more applications to the Required List to increase the constraints on exporting data. This in turn results in a change in the workflow’s label, and thus the Effective Policy of the workflow. A change in a principal’s Policy results in a new Effective Policy for the workflow, and then results in modifications in the network access privileges of each component of the workflow, as shown in Figure 6.3. Multiple events result in the addition or removal of policy, and the network access privileges of each component of the workflow are modified accordingly. After computing the effective policy, a component is granted network access only if

1. All the applications listed in the Required List $R$ of the effective policy over the workflow are currently participating in the workflow, i.e. are in some list of participants $P_W$ for workflow $W$; such that $P_W \supseteq R$. 
2. The application, i.e. owning principal of the component in question is included in the *Export List* $E$ of the effective policy on the workflow.

### 6.3.2 Enforcing the network access privilege

At runtime, if a workflow component tries to access the network, Aquifer intercepts this attempt in the kernel, and checks if the calling component’s process has been granted the network access privilege. Based on this check, the calling component is allowed/denied network access.

### 6.4 Policy propagation to storage

We need to ensure that as long as data is present in the system, restrictions set on its use by its owners are always followed. Thus, if data leaves the workflow boundaries, the restrictions set on the data should propagate along with it, so that when the data is used by another workflow, these restrictions are followed by this new workflow as well.

From our study of application interactions, we have observed that for sharing data in content providers, applications share URIs to data. Also, in most cases, the data in the Content Provider for the corresponding URI is actually information about a file, since most of the data being shared is stored in individual files. For example, images, documents, etc are stored in the form of files, as it is difficult to store them directly in a sub-file structure like a database row. Thus, if we propagate policy to files, we will be able to cover most of the accesses to application-specific data. To propagate label to the sub-file level, like database rows, we can utilize taint tracking mechanisms provided by TaintDroid[15], though we limit the scope of this work to propagate policy to only file level data.
6.4.1 Protecting Files:

We protect files by labelling them with a label that protects all the data ever written to a file. That is, if a component from a workflow $W$ writes data to a file $F$, $F$ will be relabeled. The new label on $F$ would be a join of the current label on the file $F$ and the label on the workflow $W$. The label join operation from Chapter 5 ensures that the new label is at least as restrictive as both the labels used to compute it.

If a component from another workflow $W_2$ reads the file, $W_2$’s label will be replaced by a new label, which is the join over $W_2$’s previous label, and the label on the file $F$. This is to ensure that the new label on $W_2$ protects both the data present in $W_2$ and the data that is being read from the file.

6.4.2 Propagating policy through daemons:

Many applications interact with daemons, and hence if we label daemons or try to add them to workflows, many workflows will intersect, leading to label bloat. Our requirement is to only propagate a workflow’s policy to another workflow whose component is receiving the former’s data from the daemon. That is, policy is to be propagated only if a daemon reads a file belonging to one workflow, and then sends the file descriptor to a component of another workflow.

Hence, instead of propagating policy to the daemon, we propagate policy through it. If a daemon reads a file from a workflow, we associate the file descriptor with the daemon. When the daemon sends the file descriptor to a component belonging to another workflow, we perform a label join on the label of the file and the label of the workflow receiving the file descriptor, and replace the receiver’s label with the result of the join. Thus, in addition to propagating labels when a workflow component accesses a file directly, we also
propagate labels when a workflow component accesses a file through a daemon, without propagating labels to the daemon itself.
Figure 6.1: Block Diagram of the *Aquifer* System. Shows basic interaction between *Aquifer* and a UI Workflow
Figure 6.2: Workflow filter in action: Policy enforcement in visual Workflows
Figure 6.3: Setting the network access privilege for components using the Aquifer Network Access Controller.
Chapter 7

Implementation

We have implemented the Aquifer System on Android 4.0.3(ICS), though it can be easily ported to the latest version(i.e. 4.1.1, JellyBean), with some minor changes. As shown in Figure 7.1, we implement a system service, the *Aquifer Service*, along with a linux security module(LSM) for kernel level monitoring. We also expose the *Aquifer API* to applications for creating policies. The following content in this chapter describes these and some other supporting components of the *Aquifer Architecture* in sufficient detail.

7.1 The Aquifer Service

The *Aquifer Service* is a system service we implemented in the Android framework. It is started as soon as android’s SystemServer starts. Most of the policy enforcement is completely enforced, or at least triggered, inside the *Aquifer Service*. It also maintains information on workflows, and applications use methods exposed by this service in order to modify their policies inside their workflow’s label.
7.1.1 Maintaining information on workflows

We implement a workflow abstraction, which contains a set of policies set by applications whose components are a part of the workflow. The Aquifer Service stores information related to workflows, and also keeps a track of the security label for each workflow. The

![Software Stack of the Android Operating System patched with Aquifer.](image)

Figure 7.1: Software Stack of the Android Operating System patched with Aquifer.
workflow → workflow label mapping is stored in a HashMap for efficiency. The service also stores a list of current participants in the workflow, called the ParticipantList, and a list of components that have been a part of the workflow at some point of time, the HistoryList. A component that enters the workflow is added to the ParticipantList and HistoryList both, but a component that leaves the workflow is only removed from the ParticipantList. These lists aid in implementing policies that require the current and past states of the workflow.

Additionally, the service exposes the getPolicy() and addPolicy(Policy policy) methods, for applications to retrieve, add and modify their policies in the workflow’s security label.

7.1.2 Workflow Filter using Aquifer Service

We modify Android’s ActivityManagerService to call the Aquifer Service after the Intent Resolver resolves the list of potential recipients for an intent. The Aquifer Service retrieves the security label on the current workflow, lets call it $W_{current}$, whose component initiated the intent. Then, it computes the effective policy of the workflow from the label, and retrieves the $WorkflowFilter(F)$ (described earlier in Chapter 5) from the effective policy. Android’s Intent API allows retrieval of the “action” of an intent using the method getAction() on the intent object. The workflow filter is another mapping of such actions to a list of applications whose components can receive those actions. Thus, we then retrieve the list of components allowed to receive the action set on the intent. From the list of components resolved by the IntentResolver, we filter out components whose applications are not a part of the list of apps permitted to receive an intent with the particular action set on the intent. This results in a filtered list containing only those
applications which will potentially not cause any access control failures if they try to perform the particular “action” described in the intent.

7.1.3 Adding and removing components from a workflow

We represent a workflow component by its process, as enforcement in the kernel requires process-level granularity. Hence, we create a new process for a visual component, i.e. an activity, for every new instance created, which helps us prevent intersections between workflows, often leading to label bloat, in situations where multiple instances of a component are active in separate workflows.

We modify the ActivityManagerService to notify the Aquifer Service every time an activity is started in a new process. The Aquifer Service retrieves the current workflow, and adds this process as a participant to the new workflow. The process name, process id, package name, and the resultcode is stored for every participant. The resultcode is 1 if the activity was started with a call that returns a result to the caller, else it is 0. We then compute the effective policy on the workflow. Based on the Export List(E) and Required List(R) components of the policy, the Aquifer Service sets the network access privilege for a component by making a call to the Aquifer device in the kernel.

An activity is presumed to have finished when its state changes to “DESTROYED”. Such activities’ processes are cleaned up by the system, and application code in these activities cannot execute in the “Stopped” or “Destroyed” states, as seen in the Figure [figure from android.tex] in Chapter 2. The Aquifer Service receives a notification in the form of a method call whenever an activity reaches the “Destroyed” state. We use this notification to maintain the most current state of the workflow. If the activity’s resultcode is 1, it means that it has returned data to its predecessor in the workflow, i.e. its data
will persist in the workflow even after it is cleaned up. In this case, we do not remove the policy set by the activity’s application from the workflow. Else, if the resultcode is 0, i.e. if it does not inject any data into the part of the workflow before it, we mark the policy set by the activity’s application for removal. If there is not other activity from the same application in the current workflow, we remove the application’s policy from the workflow’s security label. Finally, we remove the activity from the ParticipantList and the HistoryList, hence restoring the workflow to its proper state.

7.1.4 Setting the Network Access Policy

Aquifer sets the network access policy for workflow components in the kernel. This policy is enforced by our LSM in the kernel when a component’s process tries to access the network. Every time the label on a workflow changes, the effective policy on the label changes. This change can be in the form of additional restrictions placed on the workflow, barring even more applications from network access; or it may also allow network access for applications that were initially denied it in the previous state of the policy. Thus, the Aquifer Service calls the Aquifer Driver in the kernel to set the network policy for each component, following a modification to the workflow label.

7.1.5 Maintaining labels on Files

The Aquifer Service is notified by the Aquifer LSM in the kernel whenever an activity reads/ writes a file. The service stores a mapping of each file’s inode no. with the its security label. Whenever a workflow component reads a file, Aquifer performs a label join operation on the label of the file and the workflow, and sets the result as the new label of the workflow. When a component writes to a file, the result of the join is stored
as the label of the file instead. In case an activity has both read and write access to a file, the result of the join is stored as the new label of both the workflow and the file.

### 7.1.6 Adding Application Policies

Applications can add their policies to workflows through their components that participate in workflows. The `addPolicy(Policy policy)` method allows an application to replace its current policy in the workflow’s label with the new policy “policy”. *Aquifer* gets the pid of the calling component using the function `getCallingPid` exported by the *Binder* class, obtains the process name of the caller from the *ActivityManagerService*, and then retrieves the workflow for the component using its processName and pid. We then add the policy to this workflow’s label. In this way, we ensure that application policy modifications remain isolated and tamperproof.

### 7.2 Aquifer Device and LSM

We have implemented the *Aquifer LSM (Linux Security Module)* to act as the reference monitor in the kernel.

Every network or file access is mediated by the *Aquifer LSM*. On detecting a network access, the LSM checks the network access privilege for the process (stored in the process control block), and allows or denies the request. When a file is accessed, the LSM sends a notification to the *Aquifer Service* in the userspace, along with the pid of the accessing process and the file’s inode no. The *Aquifer Service* then performs label propagation.

We also add an LSM hook to the Binder IPC Driver in the kernel. When a file descriptor is passed between two processes in the kernel, the *Aquifer LSM* receives a notification, and we send the File descriptor, the type of access (r/w), and the receiving process’s pid
to the *Aquifer Service* in the userspace, which then performs label propagation.

The *Aquifer Device* presents an interface for the *Aquifer Service* to interact with the LSM, and to perform various tasks in the kernel, like setting the network policies of processes.

### 7.3 The Aquifer API

We expose the *Policy* and *AquiferList* classes to applications, in order to help them form policies that can then be set on workflows. A *AquiferList* is a list of strings, and is used to build policies. The *Policy* class contains two *AquiferLists*, namely the *ExportList* and the *RequiredList*, as defined in the policy specification in Chapter 5.

The *Policy* also contains a *Filter*, which is also exposed to applications as a part of the API. The *Filter* is a HashMap with action strings as keys, and *AquiferLists* of application names as corresponding values. Application developers are also provided with member methods to make policy configuration effortless.
Chapter 8

Evaluation

8.1 Performance

The Aquifer System has been implemented for Android 4.0.3(ICS) and uses Android Kernel 3.0.8. The performance evaluation was carried out on a Samsung Galaxy Nexus(I9250) phone, with a Dual-core 1.2 GHz processor and 1GB RAM.

We implement the Network Access Privilege check in the kernel, at the same location where Android checks its INTERNET permission. We observed the performance overhead of this check to be negligible.

When the LSM operation for the file permission hook is executed in the kernel, two 32 bit integer values are sent via a socket to the userspace(hence there is no context switch). Hence, even in this case, we assume the overhead to be nominal.

On the other hand, we perform a number of operations while starting an application, from modifying workflow data structures to setting the network policies for each participant of the workflow. We tested an unmodified version of Android ICS and an Aquifer supported ICS separately for application load time, and we found that Aquifer caused
only over 3% performance overhead when compared to the unmodified ICS build.

8.1.1 Security Evaluation

As discussed in the threat model in Section 3.4, 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. In fact, by specifying which permissions an app uses, and assigning permissions to restrict app interfaces, Android application developers are already defining security policy.

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 (as described in Section 9), Aquifer merges the two workflow labels using the join (⊔) operator (Definition 8). In Section 5.2, we claimed that the join operation ensures that the resulting label is at least as restrictive as both the original labels. We now formally prove this property (i.e., we prove the safety of the Aquifer policy language).

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 a 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 at $L_2$ 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, workflow filters, and required lists $E_{1e}, E_{2e}, F_{1e}, F_{2e}, R_{1e},$ and $R_{2e}$, respectively. $L_2$ is effectively more restrictive than $L_1$, denoted $L_1 \sqsubseteq_e L_2$, 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 $\sqsubseteq_o$). Let $L_1$ and $L_2$ be workflow labels, $O$ be the owner for with respect 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 $L_1 \sqsubseteq_o L_2$, 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
Definition 11 (Restriction relation \( \sqsubseteq \)). Let \( L_1 \) and \( L_2 \) be workflow labels. \( L_2 \) is more restrictive than \( L_1 \), denoted \( L_1 \sqsubseteq L_2 \), if and only if \( L_1 \sqsubseteq_e L_2 \) and \( \forall O \in \text{owners}(L_1), L_1 \sqsubseteq_O L_2 \).

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 that creates a new label \( L_1 \sqcup L_2 \). We must show that \( L_1 \sqsubseteq L_1 \sqcup L_2 \) and \( L_2 \sqsubseteq L_1 \sqcup L_2 \).

Based on Definition 11, \( L_1 \sqsubseteq L_1 \sqcup L_2 \) iff (a) for all \( O \in \text{owners}(L_1), L_1 \sqsubseteq_O L_1 \sqcup L_2 \) and (b) \( L_1 \sqsubseteq_e L_1 \sqcup L_2 \).

Condition (a) is satisfies Definition 10 by expanding \( L_1 \sqcup L_2 \) using Definition 8, as follows. For all owners \( O \in \text{owners}(L_1) \), let \( F_1 = \text{filters}(L_1, O) \) and \( F_2 = \text{filters}(L_2, O) \), then

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

Condition (b) is satisfies Definition 9 by expanding \( L_1 \sqcup L_2 \) using Definition 8 and applying Definitions 5-7 to determine the effective policy.

Export list: for \( L_1 \), \( E_{1e} = \bigcap \text{exports}(L_1, O) \) for all \( O \in \text{owners}(L_1) \). For \( L_1 \sqcup L_2 \), \( E_{12e} = \bigcap (\text{exports}(L_1, O) \cap \text{exports}(L_2, O)) \) for all \( O \in (\text{owners}(L_1) \cup \text{owners}(L_2)) \). 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 5.2) 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 $L_1 \sqcup L_2$, $R_{12e} = \bigcup (requires(L_1, O) \cup requires(L_2, O))$ 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 $L_1 \sqcup L_2$,

$$F_{12e} = \{(s, T) \mid s \in \bigcup (actions(F_1) \cup actions(F_2))$$

and $T = \bigcap (targets(F_1, s) \cap targets(F_2, s))$, \forall F_1 \in filters(L_1, O), \forall F_2 \in filters(L_2, O), \forall O \in (owners(L_1) \cup owners(L_2))\}$$

Definition 9 first requires showing that $actions(F_{1e}) \subseteq actions(F_{12e})$. 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 $\forall s \in actions(F_{1e}), targets(F_{1e}) \subseteq targets(F_{12e})$. 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 $L_1 \sqsubseteq L_1 \sqcup L_2$. The proof that $L_2 \sqsubseteq L_1 \sqcup L_2$ follows similarly and is not shown for brevity. □
8.1.2 Case Study: K-9 Mail

To demonstrate how Aquifer works in practice, we performed a case study using K-9 Mail. K-9 Mail is an open source fork of the original Email client in the Android Open Source Project (AOSP). Our case study also used the open source PDFView application, which we modified to include malicious functionality. Our modifications of PDFView include 1) sending strings from a PDF file to a network server, and 2) saving version of a PDF file, and then on a later invocation of PDFView, opening the saved file and sending strings from it to a network server.

K-9 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-9 Mail creates an intent message with the implicit address ACTION_VIEW and the data type set to application/pdf. When K-9 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. Note that this required no modifications to be compatible with Aquifer. When we viewed contract.pdf, our modified PDFView successfully exported strings from the PDF as designed.

We then modified K-9 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 at-
Attachment is treated as confidential, and potentially a method to declassify an attachment in the event of false labeling. Our second modification was set the owner policy for the UI workflow before a confidential attachment is viewed. For this we used the following owner policy.

\[
E = \{ \text{K9Mail} \} \\
F = \{ (\text{ACTION_SEND}, \{ \text{K9Mail} \}) \} \\
R = \{
\]

This policy ensures that only K-9 Mail can export the data, and if any application in the UI workflow uses the \text{ACTION\_SEND} action string to start an activity, only K-9 Mail will be shown, filtering out other options (e.g., Facebook). Adding this policy to K-9 Mail required very few changes, as shown in Listing 8.1.

We then re-performed our previous experiment. This time, when PDFView attempted to send strings from \text{contract.pdf}, it could not reach the network. Furthermore, when PDFView saved a copy of \text{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 malicious changes) and the workflow was successfully labeled, again keeping PDFView from exporting strings from the document.
Chapter 9

Related Work

This work deals with providing information flow guarantees for application specific data in a smartphone environment. Traditional IFC(Information Flow Control) models [3, 4, 9, 35, 28, 11] information flow guarantees are effective for protecting only system data, i.e. the data and data types known to the system during its initial policy configuration. In a smartphone device, say Android, which on average has more than 50 third party applications installed, and where there are more than half a million applications in the Android Market, it is impossible for the operating system to know about the data and the security requirements of the data in each of the applications. Since the operating system cannot understand application-specific data, data types and the context of their use, traditional centralized IFC models fail to protect application-specific data in smartphone environments.

This leads us to the development of application or program centric information flow control architectures which provide Decentralized Information Flow Control(DIFC) [29, 39, 44, 45, 27, 26, 38, 34, 30], which focus on policy specification by application( hence Decentralized), and the enforcement of such policies by the operating system. Such ar-
chitectures ensure information flow guarantees for app-specific data, since applications themselves specify policies regulating access to their data. DIFC models propagate policy to each process/component, while Aquifer propagates policy over the entire UI workflow. Since traditional DIFC does not keep a track of a workflow, it cannot take advantage of the state of the current workflow like Aquifer does. For example, policies that are influenced by the current or past state of the workflow cannot be enforced in a traditional DIFC architecture, nor can “workflow filters” be utilized to reduce policy violations. Additionally, these architectures do not differentiate between user interface components and background components while propagating policies, and hence cannot use the context derived from user actions in policy design.

Most work in the area of transitive information flow in smartphones is concentrated on preventing privilege escalation and preserving integrity[12, 8, 20, 6, 10], and mostly in system applications. Moreover, these works are based on Android permissions, which are opaque strings, hence lack context, and do not have sufficient granularity (i.e., many permissions can be used for both read and write access), and thus cannot be correctly used for providing secrecy or integrity guarantees. Additionally, unlike traditional privilege escalation prevention mechanisms, Aquifer can prevent data leakage through collusion, as it propagates policy from one workflow to another, both through daemons and storage.
Chapter 10

Discussion

The Aquifer architecture provides protection to application-specific data that is a part of a UI workflow. If this data is written to a file, the constraints on the use of the data propagate to the file, to be applied to later uses of the data.

We label data at the granularity of files. Additional knowledge about the flow of data is necessary to be able to extend this labeling to the granularity of sub-file data structures, database rows for example. In order to label a database row, it is necessary to obtain identifying information about the row from the library which implements the database, and at an even higher level, the Content Provider that uses the API exposed by the database library. Both the database library and the content provider abstraction can be implemented by applications themselves, hence may not be trustworthy enough to provide correct information about a database row.

In order to use the Content Provider abstraction for getting information about a database row without compromising the integrity of the information, a taint tracking mechanism like [15, 21, 7, 43] can be utilized to detect if the Content Provider modifies or caches any values received from the database library. Such an approach would lead to
a large performance overhead, and a considerable number of false positives. Additionally, it would still imply trusting the database library used by the application. One more approach would be to trust only the database library provided by default in Android, and to treat databases operations by application specific libraries as ordinary file accesses.

As the concept of “Workflow based information flow control”, deals with propagating restrictions on the data to storage, and through the storage to another workflow that reads the data, labeling files serves the purpose. In order to further reduce label bloat, we could label content provider rows by using a stripped down taint tracking mechanism, but that is beyond the scope of this work.

We have also observed instances of applications which are now working hard to get the context of the application’s use of the data. For example, Evernote lets users categorize their notes, and can infer the user’s context from categories. We provide such apps with a way to utilize this context knowledge. Future work may also deal with specifying correct policies for the UI context.
Chapter 11

Conclusion

d this work demonstrates how information flow control can be modelled along the application workflow in order to achieve additional control over the flow of application specific data.

The Aquifer architecture provides an application with the ability to specify restrictions on the use of its data, and enforces these restriction throughout the existence of the data. We show that the Aquifer policy design has the expressibility required in order to effectively state many interesting secrecy requirements of application specific data.

Using Aquifer, we show how applications utilize the valuable context derived from the user interface, and form the policy which reflects the user’s intentions and security requirements. The policy is successfully propagated to other workflows that access the data through daemons as well as storage.

We demonstrate how access control policy violations can be reduced by modifying the options presented to the user. Finally, we also show how Aquifer secures information flow for applications while incurring minimal performance overhead.
REFERENCES


[22] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren’t the Droids You’re Looking For: Retrofitting Android to Protect Data from Imperious Applications. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2011.


