Preventing Accidental Data Disclosure in Modern Operating Systems

Adwait Nadkarni
North Carolina State University
Raleigh, North Carolina, USA
anadkarni@ncsu.edu

William Enck
North Carolina State University
Raleigh, North Carolina, USA
enck@cs.ncsu.edu

ABSTRACT

Modern OSes such as Android, iOS, and Windows 8 have changed the way consumers interact with computing devices. Tasks are often completed by stringing together a collection of purpose-specific user applications (e.g., a barcode reader, a social networking app, a document viewer). As users direct this workflow between applications, it is difficult to predict the consequence of each step. Poor selection may result in accidental information disclosure when the target application unknowingly uses cloud services. This paper presents Aquifer as a policy framework and system for preventing accidental information disclosure in modern operating systems. In Aquifer, application developers define secrecy restrictions that protect the entire user interface workflow defining the user task. In doing so, Aquifer provides protection beyond simple permission checks and allows applications to retain control of data even after it is shared.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls, information flow controls*

Keywords

OS security; access control; information flow control

1. INTRODUCTION

Operating system security architectures are currently undergoing a fundamental change. Modern OSes [32, 42], such as Android, iOS, and Windows 8, take the suggestion of decades of security research [45, 22, 35, 14] and run each application as a unique security principal. While having finer-grained security principals prevents many obvious attacks, complete sandboxing [19] 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 [39], GLBA [38], PCI DSS [30], and IRS 1075 [40].

Preventing accidental disclosure is not as simple as restricting the set of applications an application with sensitive data can interact with (e.g., Saint [29]). A trusted application receiving data might share that data with another application that has unexpected disclosure. Hence, in a 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 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 paper, 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 sensi-

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516677.

tive data, thereby addressing the data intermediary problem for these large data objects.

This paper 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.

The remainder of this paper proceeds as follows. Section 2 provides a use case and problem definition. Section 3 overviews our approach. Section 4 formally defines the Aquifer policy logic. Section 5 describes the Aquifer system design. Section 6 details the implementation. Section 7 evaluates Aquifer's policy compatibility and performance. Section 8 discusses limitations. Section 9 overviews related work. Section 10 concludes.

2. MOTIVATION AND PROBLEM

Modern operating systems such as Android, iOS, 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 2.4) 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 paper, 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.

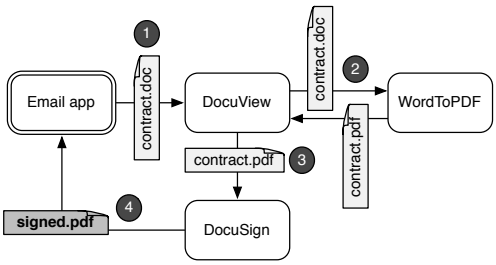


Figure 1: Document signing use case with four apps. A confidential contract received via Email is 1) read in a viewer, 2) converted to PDF, 3) embedded with a written signature, and 4) Emailed back to the sender.

2.1 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 1.

2.2 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 [39], GLBA [38], PCI DSS [30], and IRS 1075 [40]). 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 paper, 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 paper can form the basis of a system to defend against this stronger adversary.

2.3 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 paper 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.

2.4 Background: Android

Android runs a Linux kernel, but defines its own application runtime environment. 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*. Activity components define the application's user interface. 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 act as database daemons, and broadcast receiver components listen for messages.

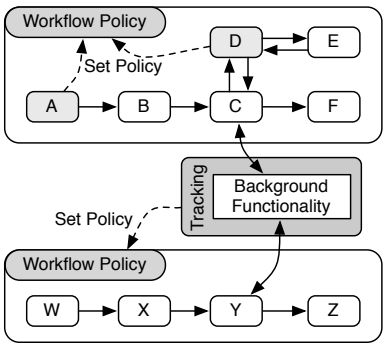


Figure 2: Aquifer policy abstraction

Android's *binder* framework 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.

3. 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 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 2, Aquifer

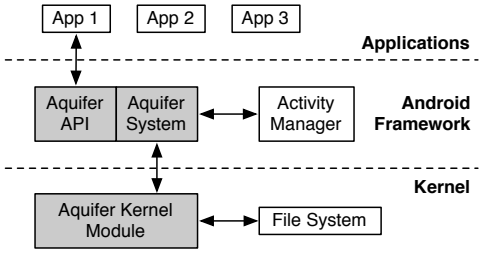


Figure 3: Aquifer architecture for Android

allows applications on the branch to contribute to secrecy restrictions (e.g., UI screen D).

Figure 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) [27]. 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 [23]. 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 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 [13] and CleanOS [37]. However, the primary focus of this paper 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 5.3).

4. 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.

4.1 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 3, 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 2, 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.

4.2 Policy Logic

Aquifer formalizes the export, required, and filter policy types into a logic. Our logic is motivated by the decentralized label model (DLM) [27]. We chose DLM over other DIFC logics [41, 47, 48, 25, 24, 36] 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 apps.

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, $actions(F)$ returns the set of all action strings in F. Similarly, for a filter F and an action string s, $targets(F, s)$ 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 $targets(F, s) = \emptyset$. If this occurs, Aquifer simply removes s from $actions(F)$, 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 $owners(L)$, 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, 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 $\mathcal{A}$, and the set of all possible action strings be $\mathcal{S}$. For each owner O_i, $exports(L, O_i)$ returns E_i, unless $O_i \notin owners(L)$ or $E_i = \emptyset$, in which case $exports(L, O_i)$ returns $\mathcal{A}$. Semantically, this means O_i does not have any export restrictions. Similarly, for each owner O_i, $filters(L, O_i)$ returns F_i, unless $O_i \notin owners(L)$ or $F_i = \emptyset$, in which case it returns $\{(s, \mathcal{A}) | \forall s \in \mathcal{S}\}$. In contrast, for each owner O_i, $requires(L, O_i)$ 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 $E_e = \bigcap exports(L, O), \forall O \in owners(L)$.

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

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, $F_e = \{(s_i, T_i) \mid s_i \in \bigcup actions(F)$ and $T_i = \bigcap targets(F, s_i), \forall F \in filters(L, O), \forall O \in owners(L)\}$.

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 [27].

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:

$$
\begin{aligned}
owners(L) &= owners(L_1) \cup owners(L_2) \\
exports(L, O) &= exports(L_1, O) \cap exports(L_2, O) \\
requires(L, O) &= requires(L_1, O) \cup requires(L_2, O) \\
filters(L, O) &= \{(s_i, T_i) | s_i \in actions(F_1) \cup actions(F_2), \\
&\qquad T_i = targets(s_i, F_1) \cap targets(s_i, F_2), \\
&\qquad \text{where } F_1 = filters(L_1, O), \\
&\qquad F_2 = filters(L_2, O)\}
\end{aligned}
$$

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. 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.

5.1 Managing UI Workflows

As described in Section 3, 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 new 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_e), and 2) W_R, a list of metadata for currently "running" UI screens (for effective export list E_e). 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 $L_1 \sqcup L_2$ 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 *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 *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_V 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.

5.2 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 list R_e, and effective workflow filter F_e.

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:

$$(E_e = \emptyset \vee app(p) \in E_e) \wedge (\forall r \in R_e, r \in W_V)$$

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 neces-

sary 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.

5.3 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., TaintDroid [13] and CleanOS [37]). 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 paper 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 $L_W \sqcup L_F$.

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.

6. 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 Activity-Manager 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 also 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.

7. EVALUATION

We now evaluate Aquifer by accessing 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.

7.1 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

Table 1: Application Survey Results

Characteristic	Number of Apps
Data sources	85 (17%)
Data intermediaries	140 (28%)
Value from Export Policy	70 (14%)
Value from Regulate Policy	78 (15.6%)

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 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.

7.2 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 ($\sqcup$) operator (Definition 8). Section 4 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 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.

Table 2: Microbenchmark Results

Benchmark	Android	Aquifer	Overhead
App load	188.49±5.36 ms	192.07±6.30 ms	1.9%
App filter	194.12±7.91 ms	195.22±7.52 ms	0.55%
Net access	108.60±6.48 ms	109.64±6.31 ms	0.53%
Policy change	-	1.98±1.27 ms	-
File Read (1MB)	4.76±0.09 ms	5.23±0.22 ms	9.87%
File Write (1MB)	23.89±0.45 ms	25.44±0.86 ms	6.49%

7.3 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 2.

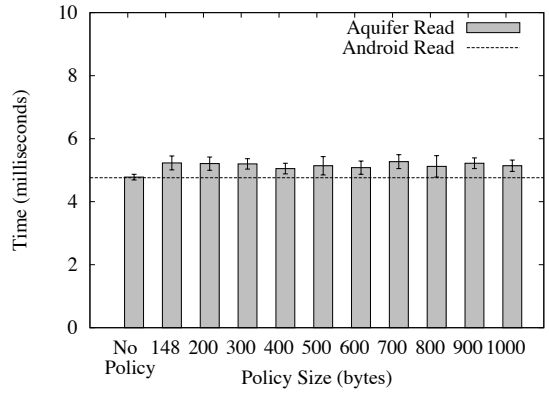
App load time: Aquifer initializes its UI workflow structures when the first application is loaded. This consists of creating a new label and data structures for W_V and W_R to maintain the workflow state. We compared the time to start the first application of a UI workflow in Aquifer to a baseline application 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.

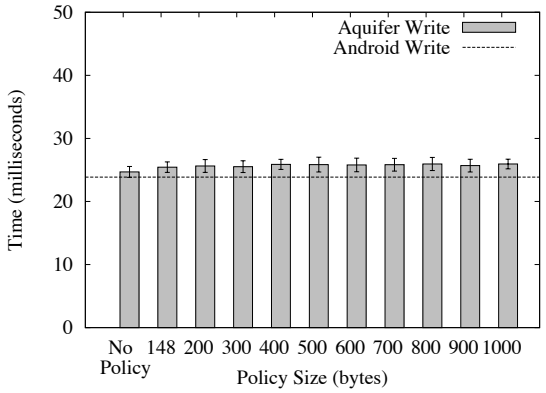
Network access check: Aquifer places a hook in the kernel that is called every time a process attempts to access the network. For this experiment, we created an application with an activity component that attempts to access the network repeatedly. Since Android already performs a similar check to enforce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

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



(a) File Read



(b) File Write

Figure 4: Aquifer File Label Propagation Time. Error bars indicate 95% confidence intervals

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 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 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 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 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.

```
1  AquiferList exportList = new AquiferList();
2  exportList.add(this.getPackageName());
3
4  AquiferFilter filter = new AquiferFilter();
5  filter.addTarget(android.intent.ACTION_SEND, this.
     getPackageName());
6
7  AquiferPolicy policy = new AquiferPolicy();
8  policy.setExportList(exportList);
9  policy.setFilter(filter);
10
11 IAquiferService aquifer = IAquiferService.Stub.
     asInterface(ServiceManager.getService("Aquifer"));
12 aquifer.addPolicy(policy);
```

Listing 1: Aquifer policy modifications to K-9E Mail

7.4 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.

7.4.1 Case Study 1 (Confidential PDF)

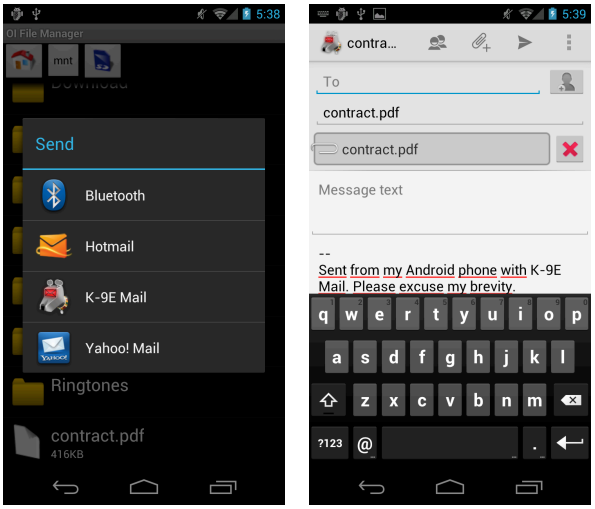
K-9 Mail is an open source fork of the original Email client in the Android Open Source Project (AOSP). We modified K-9 Mail to create K-9E Mail, an enterprise email client for use by the employees of a fictional enterprise. We also used the open source PDFView application, which we modified to emulate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



(a) Without Filter Policy (b) With Filter Policy

Figure 5: Aquifer Workflow Filter, a) Without policy and b) With Policy that allows only K-9E Mail

We used the following owner policy.

$$E = \{\texttt{K9EMail}\}$$
$$R = \{\}$$
$$F = \{(\texttt{ACTION\_SEND}, \{\texttt{K9EMail}\})\}$$

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. Adding this policy to K-9 Mail required very few changes, as shown in Listing 1.

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.

7.4.2 *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 documents 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 *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.

$$E = \{\texttt{ALL}\}$$
$$R = \{\texttt{OI File Manager}\}$$
$$F = \{\}$$

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.

7.4.3 *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.

8. 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.

9. RELATED WORK

Modern OSes, such as Android, iOS, and Windows 8, take the suggestion of decades of security research [45, 22, 35, 14] and run each application as a unique security principal. In these systems, security policy is defined with respect to permissions, which are granted to apps and restrict access to APIs and other applications. Research has criticized Android's permission framework for being both too coarse grained [3, 16] and too confusing for users [17]. Researchers have built enhanced security frameworks around Android permissions [15, 29, 7, 28, 6], some of which ease policy specification, while others make it more complicated.

Ultimately, permissions lack transitive semantics, which make them insufficient to express the security goals of mod-

ern OSes, as demonstrated by Android permission privilege escalation attacks [9, 20]. IPC Inspection [18] adds transitivity by reducing app permissions at runtime, similar to Biba low watermark [5]. Unfortunately, this requires apps to request extra permissions to operate, resulting in permission bloat. In contrast, Quire [11] adds IPC provenance records to help developers prevent confused deputy attacks. From the secrecy perspective, TaintDroid [13] and AppFence [21] use dynamic taint analysis to determine when privacy sensitive values such as location and phone identifiers are sent to the network. However, they lack the necessary policy semantics to address the data intermediary problem.

Traditionally, OS protection systems provide transitive protection semantics using information flow control (IFC) [10, 4, 5]. IFC labels subjects and objects, and uses a lattice to define a relation between the labels. Original IFC systems (e.g., MLS [4]) use a central definition of security labels, which does not meet the needs of software that defines new types of information (i.e., apps in modern operating systems). Myers and Liskov [27] defined a decentralized label model (DLM) that has formed the policy model for several decentralized information flow control (DIFC) operating systems (e.g., Asbestos [41], HiStar [47], Flume [25, 24], Laminar [33], and Fabric [26, 2]). DIFC allows applications to define their own label types.

Aquifer's policy is based on DIFC semantics, but it is optimized for the specific needs of the data intermediary problem. Traditional DIFC regulates interaction between processes and access to data objects. In modern OSes like Android, apps are frequently purpose specific and complex tasks are performed in user-driven workflows. Strict restrictions on communication and data sharing among applications would disrupt these workflows and limit the user to using only specific applications. Therefore, we relax DIFC constraints to enable greater inter-app data sharing, providing applications with a mechanism to control exfiltration of their data off the device instead.

Previous systems have controlled accidental data disclosure. Compartmented Mode Workstations (CMW) [8] assign and propagate sensitivity labels to data objects and processes (e.g. a screenshot has the label of the highest secrecy level of the data windows captured in it). A user action that leads to a flow from a high secrecy to a low secrecy level is met with a dialog box, confirming if the action was intended. Trusted Window Systems [31, 34] prevent accidental cut-and-paste actions by the user from a high-secrecy document to a low-secrecy document altogether.

Other factors leading to accidental data exposure, such as missing access control checks and poor programming have also motivated research. Resin [46] allows the programmer to configure application level data assertions to prevent accidental information leaks in Web applications. Resin is designed to help programmers detect vulnerabilities and bugs in their own applications, but does not provide Aquifer-like information flow guarantees in a multi-application environment. Liquid Machines [44] provides enterprise support for policy compliant remote content access using encryption. The use of encryption may restrict the user from using third party applications.

Finally, we are not the first to consider the challenges of modern operating systems. The ServiceOS project at Microsoft Research, which includes MashupOS [42] and Gazelle [43], considers similar problems, but focuses on Web browsers. Also under this umbrella project is Access Control Gadgets (ACG) [32], 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 [12] and DarpaBrowser [1] to grant a process access to a file based on the user's natural file selection process. Along these lines, Aquifer infers data access based on the UI workflow as the user performs a task.

10. CONCLUSION

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.

Acknowledgements

This work was supported in part by an NSA Science of Security Lablet grant at North Carolina State University and NSF grants CNS-1222680 and CNS-1253346. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We would also like to thank Patrick McDaniel, Patrick Traynor, Kevin Butler, Tsung-Hsuan Ho, Ashwin Shashidharan, Vasant Tendulkar, and the anonymous reviewers for their valuable feedback during the writing of this paper.

11. REFERENCES

[1] A Capability Based Client: The DarpaBrowser. `http://www.combex.com/papers/darpa-report/html/`.

[2] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing Mobile Code Securely With Information Flow Control. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[3] D. Barrera, H. G. Kayacik, P. C. van Oorshot, and A. Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2010.

[4] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, Apr. 1977.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Toward Taming Privilege-Escalation Attacks on Android. In *Proceedings of Network and Distributed System Security Symposium*, 2012.

[7] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings Information Security Conference*, 2010.

[8] P. T. Cummings, D. A. Fullam, M. J. Goldstein, M. J. Gosselin, J. Picciotto, J. P. Woodward, and J. Wynn. Compartimented Mode Workstation: Results Through

Prototyping. In *In the IEEE Symposium on Security and Privacy*. IEEE, 1987.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)*, Oct. 2010.

[10] D. E. Denning. A Lattice Model of Secure Information Flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[12] E and CapDesk. `http://www.combex.com/tech/edesk.html`.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[14] W. Enck, P. McDaniel, and T. Jaeger. PinUP: Pinning User Files to Known Applications. In *Proceedings of Annual Computer Security Applications Conference*, 2008.

[15] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[16] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[17] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security*, 2012.

[18] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of USENIX Security Symposium*, 2011.

[19] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the USENIX Security Symposium*, 1996.

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2012.

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. 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.

[22] S. Ioannidis, S. Bellovin, and J. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proceedings of ACM SIGOPS European workshop*, 2002.

[23] P. F. Klemperer, Y. Liang, M. L. Mazurek, M. Sleeper, B. Ur, L. Baur, L. F. Cranor, N. Gupta, and M. K. Reiter. Tag, You Can See It! Using Tags for Access Control in Photo Sharing. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2012.

[24] M. Krohn and E. Tromer. Noninterference for a Practical DIFC-Based Operating System. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[25] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2007.

[26] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1997.

[28] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of ASIACCS*, 2010.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009.

[30] Payment Card Industry (PCI). Data Security Standard: Requirements and Security Assessment Procedures, Version 2.0, Oct. 2010. Available at `https://www.pcisecuritystandards.org/security_standards/documents.php`.

[31] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*. IEEE, 1991.

[32] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[33] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proc. of the Conference on Programming Language Design and Implementation*, 2009.

[34] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium*, 2004.

[35] P. Snowberger and D. Thain. Sub-Identities: Towards Operating System Support for Distributed System Security. Technical Report 2005-18, University of Notre Dame, Department of Computer Science and Engineering, 2005.

[36] D. Stefan, A. Russo, D. Mazieres, and J. C. Mitchell. Disjunctive Category Labels. In *Proc. of NordSec*, 2011.

[37] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposiure with Idle Eviction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[38] US Congress. Gramm-Leach-Bliley Act, Finiancial Privacy Rule. 15 USC §6801-§6809, Nov. 1999. Available at `http://www.law.cornell.edu/uscode/usc_sup_01_15_10_94_20_I.html`.

[39] US Congress. Health Insurance Portability and Accountability Act of 1996, Privacy Rule. 45 CFR 164, Aug. 2002. Available at `http://www.access.gpo.gov/nara/cfr/waisidx_07/45cfr164_07.html`.

[40] US Internal Revenue Service (IRS). Publication 1075: Safeguards for Protecting Federal Tax Returns and Return Information, 2010. Available at `http://www.irs.gov/pub/irs-pdf/p1075.pdf`.

[41] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)*, 25(4), December 2007.

[42] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[43] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principle OS Construction of the Gazelle Web Browser. In *Proceedings of the USENIX Security Symposium*, 2009.

[44] B. Week. Company Overview of Liquid Machines, Inc. `http://investing.businessweek.com/research/stocks/private/snapshot.asp?privcapId=3079632`.

[45] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: An Access Control List Approach to Anti-viral Security. In *Proceedings of the 13th National Computer Security Conference*, 1990.

[46] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[47] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[48] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing Distributed Systems with Information Flow Control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2008.

APPENDIX

A. AQUIFER POLICY SAFETY PROOF

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 $L_1 \sqsubseteq_e L_2$, if and only if:

$$E_{1e} \supseteq E_{2e}$$
$$R_{1e} \subseteq R_{2e}$$
$$actions(F_{1e}) \subseteq actions(F_{2e})$$
$$\forall s \in actions(F_{1e}), targets(F_{1e}, s) \supseteq targets(F_{2e}, s)$$

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 which the relation is evaluated, $F_1 = filters(L_1, O)$, and $F_2 = filters(L_2, O)$. L_2 is more restrictive than L_1 for owner O, denoted $L_1 \sqsubseteq_O L_2$, if and only if:

$$exports(L_1, O) \supseteq exports(L_2, O)$$
$$requires(L_1, O) \subseteq requires(L_2, O)$$
$$actions(F_1) \subseteq actions(F_2)$$
$$\forall s \in actions(F_1), targets(F_1, s) \supseteq targets(F_2, s)$$

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

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* $\forall O \in owners(L_1), L_1 \sqsubseteq_O L_2$.

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 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 $O \in owners(L_1), L_1 \sqsubseteq_O L_1 \sqcup L_2$ 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 $O \in owners(L_1)$, 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 satisfies Definition 9 by expanding $L_1 \sqcup L_2$ 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 $L_1 \sqcup L_2$, $E_{12e} = \bigcap(exports(L_1, O) \cap exports(L_2, O))$ 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 4) 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, $R_{1e} \subseteq R_{1e2}$, 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 $L_1 \sqcup L_2$,

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

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. □