Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems

Riley Spahn and Jonathan Bell, *Columbia University;*
Michael Lee, *The University of Texas at Austin;*
Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser, *Columbia University*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/spahn

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the
11th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems

Riley Spahn, Jonathan Bell, Michael Z. Lee*, Sravan Bhamidipati, Roxana Geambasu, *and* Gail Kaiser
*Columbia University, *The University of Texas at Austin*

Abstract

Support for fine-grained data management has all but disappeared from modern operating systems such as Android and iOS. Instead, we must rely on each individual application to manage our data properly – e.g., to delete our emails, documents, and photos in full upon request; to not collect more data than required for its function; and to back up our data to reliable backends. Yet, research studies and media articles constantly remind us of the poor data management practices applied by our applications. We have developed Pebbles, a fine-grained data management system that enables management at a powerful new level of abstraction: *application-level data objects*, such as emails, documents, notes, notebooks, bank accounts, etc. The key contribution is Pebbles's ability to discover such high-level objects in arbitrary applications *without* requiring any input from or modifications to these applications. Intuitively, it seems impossible for an OS-level service to understand object structures in unmodified applications, however we observe that the high-level storage abstractions embedded in modern OSes – relational databases and object-relational mappers – bear significant structural information that makes object recognition possible and accurate.

1 Introduction

Despite recent high-profile failures in applications' management of our data [2], in the absence of system-level support for fine-grained data organization, we are forced to entrust them with our data. When users perform day-to-day data management activities – deleting individual emails, identifying specific data that was viewed, or sharing pictures – they are forced to rely on applications to behave properly. Yet, a 2010 study of 30 popular Android applications showed that 20 leaked sensitive data, such as contacts or locations [11]. Our own study of deletion practices within mobile apps, described later in this paper, revealed that 18 of 50 popular Android applications left information behind instead of deleting it. Notably, we found that until 2011, Android's default email application left behind the attachments of deleted emails while deleting the messages themselves.

Although a plethora of system-level data management tools exist – including encrypted file systems [14, 16], deniable file systems [42], auditing file systems [12], or assured delete systems [28] – these tools operate at a single level of abstraction: *files*. Without a one-to-one mapping between user-relevant objects (for example, individual email messages in a mail client or documents in a word processor) and files, such systems provide poor granularity, preventing end-users from protecting individual objects that matter to them.

Consider Android's default email application: it stores each email's contents and to/from/subject fields as several rows in a SQLite database (all emails are stored in the same DB, which is itself stored as a single file), attachments as files, and cached renderings of messages in different files. Such complex object-to-file mappings are typical in Android, as our large-scale measurement study of Android storage patterns shows (§3). Moreover, others have observed complex storage layouts in other OSes, such as OSX, where researchers have concluded that "a file is not a file" but a complex structure with complex access patterns [18].

Given the complexity of these object-to-file mappings, we ask: is it possible for system-level tools to support management and protection at the granularity of user-relevant objects? Intuitively, this would require developers to specify the structure of their applications' persisted data to the operating system. Nevertheless, we observe that the high level storage abstractions included and predominant in today's operating systems – the SQLite relational database in Android and the CoreData object-relational mapper in iOS – bear sufficient structural information to recover these user-relevant data objects from unmodified applications.

We call these objects *logical data objects* (LDO), examples of which include an email (including its to, from, subject, body, attachments and any other related information); a mailbox including all emails in it; a bank account in a personal finance application; etc. We present *Pebbles*, a system that exposes LDOs to protection tools, without introducing any new programming models or interfaces, which can be prone to programmer error, slow adoption, or incompatibility with legacy applications.

We implemented Pebbles and several new protection tools based on it on the Android platform. Each of these tools provides protection at the LDO level, leveraging Pebbles to greatly simplify their development. Using Pebbles tools, users can mark objects from their existing applications to verify their proper deletion, protect their access from other applications, and back them up to the clouds they trust.

In a study of 50 popular Android applications, we found Pebbles to be highly effective in automatically identifying LDOs. Across these apps, object recognition recall was 91% and precision was 97%. In other words, in 91% of the cases, there was no leakage of data from user-visible objects to LDOs, and in 97% of the cases, there was no over-inclusion of extra data beyond user expectation in LDOs. Pebbles relies on several key assumptions based on common practices. Many of the cases in which Pebbles had poor accuracy, it could have been addressed had the developers followed these common practices.

Overall, this work makes the following contributions:

1. A study of over 470,000 Android apps, analyzing, for the first time at scale, the storage abstractions in common use today (§3). Our results suggest major differences compared to traditional storage abstractions, which render file-level data management ineffective while creating untapped opportunities for object-level data management.
2. The first design and implementation of a persistent data object recognition system that requires no app changes (§4 and §6). Our design taps into the opportunities observed from our large-scale Android app study. We make our code available from `https://systems.cs.columbia.edu/projects/os-abstractions`.
3. Four protection tools implemented atop Pebbles, demonstrating the power and value of application-level objects to protection tools (§5).
4. An evaluation of LDO construction accuracy with Pebbles over 50 popular applications from Google Play, showing it to be effective in practice (§7) and underscoring its well-defined failure modes (§8).

2 Motivation and Goals

We begin by presenting a set of example scenarios that highlight the need for fine-grained data management support within modern OSes.

2.1 Example Scenarios

Scenario 1: Object Deletion: Ann, an investigative journalist, has received an extremely sensitive email on her phone with an attachment that identifies her sources. To protect her sources, Ann does her due diligence by deleting the email immediately after reading its contents and restarting her phone to clean up any traces left in memory. Her phone is already configured with an assured-delete file system [28] that deletes data promptly upon request. Worried that the application might have created a copy of her data without her knowledge or control, she wonders: *Is there any remnant of that email left anywhere on the phone?* She is disappointed to realize that she has zero visibility into the data stored on her device. Weeks later, she learns that her fears were well-founded: the email app she is using contains a bug that leaves attachments intact when an email is deleted.

Scenario 2: Object Access Auditing: Bob, a financial auditor, uses his phone for all interactions with client data while on field engagements. Recently, Bob's device was stolen. Fearing that his fingerprint unlock might not withstand motivated attackers [41], Bob asked his IT admin a natural question: *Has any of my clients' data been exposed?* The admin's answer was mixed. Although activity on Bob's phone was tracked by a remote auditing file system [12], the logs show that a file, `/data/data/com.android.email/cache/7dcee8`, was accessed immediately before the phone's wipe-out. The file stores the HTML rendering of an email, but no one knows *which* email. Bob is left wondering what he should disclose to clients about the potential exposure of their data, and to *which* clients, since neither he nor the IT staff can map that file to a specific client or email.

Scenario 3: Object Access Restriction: Carla, a local politician, uses her phone to take photos for professional purposes, but she has several personal photos on it as well. She uses a cloud-based photo editor to enhance her promotional photos before posting them. Due to the coarse-grained permissions model of her Android device, she must provide this photo editor with access to all of her photos in order to use it. Carla is concerned that the photo editor may be secretly collecting all the photos from her device, including several potentially sensitive photos that could be politically compromising.

2.2 Goals and Assumptions

The above hypothetical users, along with millions of real-life users of mobile technology, have a mental model of application-level objects that is not matched by current protection tools. Ann wants to ensure that a particularly sensitive email is deleted in full, including attachments, to, from, any related caches, and other fields; Bob wants to know the sender or contents of a compromised email instead of a meaningless file name; Carla wants to protect a few of her most sensitive photos from prying applications. Traditional protection tools, such as file-based encryption, auditing, or secure deletion cannot fulfill these needs because the mapping between objects and files is application-specific and complex. The alternative, whole-disk encryption [1, 38], does not provide the flexibility that these users need.

To support such object-level data management needs, we developed Pebbles, a system that automatically reconstructs application-level logical data objects (LDOs) from unmodified applications. Pebbles exposes these LDOs to any system-wide protection tool that could benefit from understanding application-level objects. An encryption system could use LDOs to support meaningful fine-grained protection as an extra layer on top of whole-disk encryption. An auditing system could use LDOs to provide meaningful information about an accessed component. An object manager could reveal to users which parts of an object are left after deletion. And a backup system could let users choose their most sensitive objects for backup onto a trusted, self-managed server, letting the rest be backed up into the cloud.

Goals. The Pebbles design was guided by three goals:

G1: Accurate and Precise Object Recognition: Pebbles objects (LDOs) must closely match application-level persisted objects. This includes: (a) *avoiding data leaks* (if an item belongs to an LDO it must be included), and (b) *avoiding data over-inclusions* (if an item does not belong to an LDO it should not be included).

G2: Meaningful Granularity: Pebbles must recognize LDOs that are meaningful to users, such as individual emails.

G3: No New Application APIs: Pebbles must not require app developers to use new APIs; it can recommend developers to follow existing common practices but must work well even if they do not precisely follow.

Our first goal is accurate and precise object recognition (*G1*). We aim to achieve (1) good object recognition *recall* by avoiding leaks and (2) good object recognition *precision* by avoiding over-inclusions. We acknowledge that perfect recall or precision cannot be guaranteed in either an unsupervised approach or in a supervised API approach with imperfect developers, since a poorly written app could convolute data structure in a way that Pebbles cannot recover. However, we wish to formulate clearly all potential sources of leakage, to design mechanisms to address the leakages for most applications (§4.2), and to remind developers how they could avoid such leakages by following existing common practices (§8).

Related to G1, our second goal (*G2*) is to recognize relevant and meaningful LDOs. For example, in an email app, Pebbles should be able to recognize individual emails, not just coarse accounts with many emails. We note here that Pebbles identifies application-level objects that are persisted in stable storage, and we assume that those have a direct mapping onto the objects that users interact with and wish to protect.

G3 stems from our skepticism that developers will convert applications to use new security-related APIs or correctly use such APIs. However, we do expect that most developers will follow certain common practices (as evaluated in §3). Pebbles addresses this by leveraging application-level semantics already available within storage abstractions such as database schemas, XML structures, and the file system hierarchy. Pebbles also provides recommendations for developers which are rooted in already popular development practices (§8).

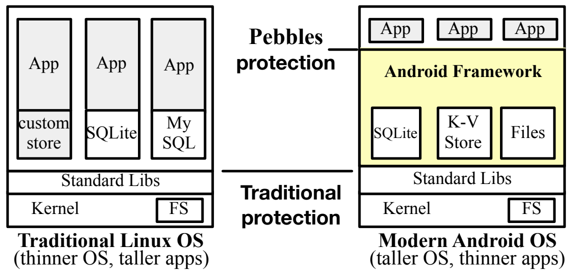


Fig. 1: **OS Storage Abstraction Evolution.** Modern OSes provide higher-level abstractions for data management, yet protection is often at the traditional file level. Pebbles, our aligns data protection with modern abstractions.

Threat Models and Assumptions. Pebbles is designed to support fine-grained data management – such as encryption, auditing, and deletion of individual emails, photos, or documents – within modern OSes. The specific threat model for a given protection tool depends on that tool's goal; however, Pebbles's mechanisms should bolster the guarantees applications can provide. In general, we assume that protection tools are *trusted* system-wide services. This is similar to assumptions made by encrypted file systems, assured-delete file systems, and other current fine-grained data management tools.

We also assume that mobile applications that create or have access to a particular object, or part thereof, will not obfuscate their data's structure or act maliciously against Pebbles. For example, they will not create their own data formats and will not willfully interfere with analysis mechanisms involved in object discovery. An application that has not yet been given access to data of a particular object, however, need not be trusted.

The scope of Pebbles is confined to those application-level objects that are persisted into a device's stable storage. We explicitly ignore attackers with access to either RAM or the underlying OS or hardware. If volatile memory protection is important, we recommend combining Pebbles with secure memory deallocation [6, 7, 15], OS buffer cleaning [10], and idle in-RAM data eviction [39] mechanisms. We also assume that secure disk scrubbing [29, 40] is deployed. In addition, while many modern applications include a cloud component, which stores or backs up data, Pebbles currently ignores that component. In the future, we plan to extend Pebbles LDOs to transcend the local and cloud environments.

While some may believe that users are incapable of dealing with fine-grained controls, we believe that there are many circumstances in which users want and are capable of handling *some* level of control, particularly for their most sensitive data. Evidence that users are capable of handling, and require, some level of control *when they feel it is important for them to do so* is available in prior studies [5, 20]. Such evidence can also be gauged

Storage Abstraction	# Apps (of 98)	Example Apps
No storage	5	Cardio Trainer
DB only	43	**CWMoney**, Amazon, BestBuy, Browser, Calendar, Contacts, ColorNotes, EverNote
FS only	3	Exchange Rates
KV only	5	Google Talk, Biorythms
DB+FS	24	**OINote**, Angry Birds, DropBox, Gallery
DB+KV	1	Twitter
FS+KV	2	Adobe Reader, Temple Run
DB+FS+KV	15	**Email**, Antivirus, Amazon Kindle, Astro File Manager, Box, EBay

(a) Use of SQLite (DB), FS, and key/value (KV) store

Storage Library	# of Apps (of 476,375)
ORMLite	6,846 (1.4%)
SQLCipher	168 (0.3%)
DB4o	116 (0.2%)
H2	16 (0.0%)
Other 4 libs combined	38 (0.0%)

(b) Third-party library use

App	Object	DB/FS Use
Email (DB+FS+KV)	Email	to/from/date in one DB table; contents in another table; attachments in FS
	Mailbox	name/server/account in one DB table; includes emails; backup in kv
	Account	address/meta data in one DB table; includes mailboxes, emails
OINote (DB+FS)	Note	title/note/tags/ in one DB table; notes exported as files in /sdcard FS
CWMoney (DB only)	Expense	name/amount in one DB table
	Category	category name in one DB table; includes expenses
	Account	name/balance in one DB table; includes categories, expenses

(c) Example object structures

Fig. 2: **Storage API Usage in 98 Android Applications.** (a) Number of apps that use the various storage abstractions in Android. Most apps use DB, but many also use FS and KV together with DB. (b) Use of eight other storage libraries among 476K free apps from Google Play. Third-party storage libraries are largely irrelevant. (c) Structure of sample objects in a few popular apps. Object structure is complex and spans multiple abstractions.

from the immense popularity of data hiding apps, such as Vault-Hide [25] and KeepSafe Vault [19], which have garnered over 10 million downloads each and let users hide data, such as photos, contacts, and SMSes.

3 Study: Android Storage Abstractions

The Pebbles design is motivated and informed by our high-level observation that storage abstractions within modern OSes are evolving in major yet unquantified ways. Fig.1 shows this evolution. Specifically, we hypothesize that the inclusion of high-level storage abstractions, such as the SQLite database in Android or the CoreData abstraction in iOS, has created a new "narrow waist" for storage abstractions that largely hides the traditional hierarchical file system abstraction. These new storage abstractions should bear sufficient structure to let us reverse engineer application-level data objects from the OS's vantage point.

In this section, we perform a simple measurement study to gauge the use of these abstractions and extract useful insights to inform our design of Pebbles. We specifically ask the following questions:

Q1 What storage abstractions do Android apps use?
Q2 How do individual apps organize their data?
Q3 How are these abstractions used?

Background. Android provides three storage abstractions [13] relevant to this paper: 1. *SQLite Database:* Stores structured data. 2. *XML-based Key/Value Store:* Stores primitive data in key/value pairs (also known as the SharedPreferences API). 3. *Files:* Stores unstructured data on the device's flash memory.

Methodology. We ran both *static* and *dynamic* experiments. Static experiments can be run at large scale but lack precision, while dynamic experiments provide precise answers but can only be run at small scale. For static experiments, we decompiled Android applications and searched their source code for imports of the storage abstractions' packages (e.g., `android.database.sqlite`). We ran large-scale, static experiments on 476,375 apps downloaded through a February 2013 crawl of Google Play [44], the main Android app market. For the dynamic experiments (over 98 apps), we installed Android apps on a Nexus S phone, manually interacted with them, and logged their accesses to the various APIs. These were some of the most popular apps, cutting across categories such as email clients, editors, banking, shopping, social, and gaming.

Results. *Q1 Answer: Apps primarily use SQLite, but use other abstractions as well.* Fig. 2(a) classifies apps according to the Android-embedded storage abstractions they use during execution. It shows that the usage of Android-provided abstractions – SQLite (denoted DB) and the key/value store (denoted KV) – eclipses the traditional file abstractions (denoted FS). Very few apps rely on the FS as their only storage abstraction (4/98). Almost half of the apps rely solely on SQLite for all of their storage needs (43/98), while almost all apps that have some local storage use SQLite (81/92). Even apps that one would consider to be primarily file-oriented (e.g., Astro File Manager, DropBox) use SQLite. A significant fraction of the apps (41/98) rely on more than one abstraction, and a notable fraction (15/98) rely on all three abstractions. This last result suggests a complex disk layout, a topic discussed further below. Overall, the most popular formations are: DB-only (43/98), DB+FS (23/98), and DB+FS+KV (15/98).

A related question is whether mobile apps use storage abstractions *other* than those provided by Android. Angry Birds, for example, stores game data and high scores in opaque binary files. We also searched the Internet for recommended Android storage options beyond those included in the OS, finding eight third-party libraries. We searched our 476K-app corpus for use of those libraries, and present the results in Fig. 2(b). None of these libraries are popular: only 2% of the apps use even one of them. Our dynamic experiments found that none of these libraries are used and provided no indication of additional libraries that we might have overlooked.

Q2 Answer: Data objects span multiple storage abstractions. Fig. 2(c) shows the structures of several logical data objects, representative of what users think and care about in various applications. It shows that objects often have complex structures that involve multiple storage abstractions. For example, Android's default email client, an example of the DB+FS+KV formation, stores various fields of the email object in two DB tables, attachments in the FS, and account recovery information in the KV. Object structure is fairly complex even for DB-only apps, such as CWMoney, a personal finance app, where a category includes metadata in one table and all expenses in another table. It thus spans multiple tables that are not linked together through explicit foreign keys. This suggests that protecting each storage abstraction separately will not work: any data protection abstraction at the end-user object level must span multiple storage abstractions.

Q3 Answer: SQLite is the hub for data management. Given this complexity, a natural question concerns how one can even begin to build some meaningful protection abstraction. Using a modified TaintDroid (a popular data flow taint tracking system for Android [11]) version, we tracked the flow of data between storage abstractions, confirming that at least 70/81 apps that use the DB use it as a *central hub* for managing their data. By central hub, we mean that data flows mostly from the DB into the FS/KV (when they are used) or is accessed using pointers from the DB; an observation that was true for 27 of the 38 apps that use FS or KV in addition to the DB. For example, many apps, including Email, use files to store caches of rendered versions of data stored in SQLite (such as the body of an email) or blobs of data that are indexed and managed through SQLite (such as the contents of pictures, videos, or email attachments).

Thus, SQLite is not just frequently used; it is the central abstraction in Android that originates or indexes much of the data stored in the other abstractions. This result is encouraging because, intuitively, relational databases bear more explicit structure.

Implications for the Pebbles Design. Overall, our results suggest that while the storage abstraction landscape is fairly complex in Android, there is sufficient uniformity to warrant constructing of a broadly applicable object system. Such a system must detect relationships between objects stored in different abstractions. The results suggest that SQLite, a relational database that bears significant inherent structure, is the predominant storage abstraction in Android. Raw files, which lack such structure, are just used for overflow storage of bulk data, such as images, videos, and attachments. Based on these insights, we construct Pebbles, the first system to recognize application-level objects within modern operating systems without application modifications.

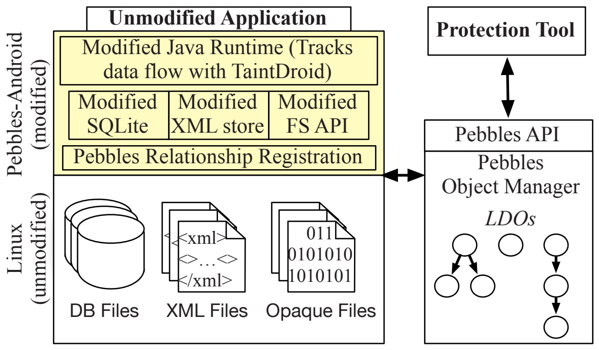


Fig. 3: **The Pebbles Architecture.** Consists of a modified Android framework and a device-wide Pebbles Object Manager. The modified framework identifies relationships between persisted data items, such as rows, XML elements, or files. The Pebbles Object Manager uses those relationships to construct an object graph; nodes map to persisted data items and edges map to relationships.

4 The Pebbles Architecture

Pebbles aims to reconstruct application-level LDOs – emails and mailboxes in an email app, saved high scores in a game, etc. – from the bits and pieces stored across the various data storage abstractions without requiring application modifications.

4.1 Overview

Fig. 3 shows the Pebbles architecture, which consists of two core components: (1) *Pebbles Android*, a modified Android framework that interposes on the various storage APIs, and (2) the *Pebbles Object Manager*, a separate device-wide entity for building object graphs and interacting with protection tools.

At the most basic level, the Pebbles Android framework understands units of storage (e.g., rows in DB, elements in XML, and files in FS) which become nodes in our object graph. The Pebbles Android framework then retrieves explicit relationships between these nodes and derives implicit relationships by tracking data flows between these units. The Pebbles Android framework registers these relationships with the Pebbles Object Manager using an internal registration API. The Pebbles Object Manager then stores these relationships, compiles a device-wide *object graph*, derives LDOs from the graph, and exports the LDOs to protection tools via the Pebbles

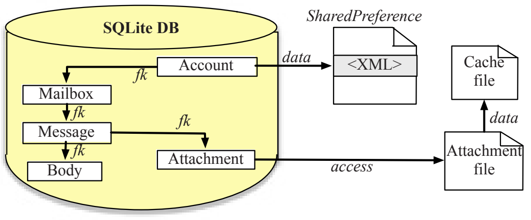


Fig. 4: **Android Email App Object Structure.** A simplified object graph for one account with one mailbox, message, and attachment. Each node represents an individual file, row, or XML element, and each edge represents a relationship. While objects can be spread across the DB, FS, and Shared Preferences, the DB remains the hub for all data.

API. LDOs are defined as follows: given a node in the graph (e.g., corresponding to a row in the `Email` table) an LDO is the transitive closure of the nodes connected to it. §7 evaluates Pebbles performance in terms of precision and recall. In the context of the graph, a failing of recall is missing nodes which should be included in a transitive closure ("leakage"); a failing of precision is including nodes which should *not* be included in a transitive closure ("over inclusion").

To provide a concrete example of the challenges faced by Pebbles, consider Fig. 4, a simplified view of how data is stored by the default Android Email application. As described previously in §3, this app stores its data across all three storage abstractions: SQLite database, SharedPreference and individual files. Although a SharedPreference is used for account recovery, and several files are used to store an attachment and a cached rendering of it, the majority of the data is stored in SQLite.

4.2 Building the Object Graph

The object graph is the center of innovation in Pebbles: it directly represents Pebbles's understanding of the structure of an app's data and lets it construct LDOs. Each file, row, and XML element is assigned a 32 bit device-wide globally-unique ID (GUID) that is stored with the data item, which are hidden from and unmodifiable by applications. For database rows, the GUID is stored as an extra column in the row's table; for XML, it is stored as an attribute of each element; and for files, it is stored in an extended attribute. When a row, element, or file is read, the data coming from it is "tainted" with its GUID and tracked in memory using a modified version of the TaintDroid taint tracking system [11].

Pebbles builds the object graph incrementally by adding new files/rows/XML elements as nodes into the graph as they are created. It also adds directed edges (called *relationships*) between nodes in the graph as they are discovered. For example, when data tainted with one GUID is written into a file/row/XML element with another GUID, a relationship is registered. All nodes and edges of the graph are registered by the modified Android framework with the Pebbles Object Manager, where they are persisted in a database. We next describe the mechanisms used to build this graph, formalized in Fig. 5.

Data flow propagation relationships: It is easy to see a strawman approach to detecting relationships between objects: when Pebbles detects that data tainted with node A's GUID is written into node B, it adds $A \leftrightarrow B$ to the object graph. This approach can capture all data flow relationships that occur within an application, regardless of the storage abstraction used. However, without precise information about the relationship between the two nodes, Pebbles is forced to assume the "worst case" scenario: that both nodes are part of the same LDO. Left unchecked, this so called taint explosion could eventually lead to all of an app's objects being included in the same LDO. Such behavior contradicts our primary goal of accurate and precise object recognition (G1). As we will see in §7.1, this naïve approach leads to unacceptably low precision (70%).

Utilizing explicit relationship information: Our next relationship detection mechanism relies on *explicit relationships* that directly communicate the programmer's view of his data structure to improve the precision. In a relational database, explicit relationships are defined in the form of foreign keys (FKs), which encode the precise relationship between two tables, based on primary keys (PKs). Interestingly, we can also extract a notion of foreign keys when relating DB rows to files: in some apps, the name of the file corresponds to the PK of the row to which it refers. Foreign keys encode the directionality of relationships, specifying for instance the difference between a "has-a" relationship and an "is-part-of" relationship. If node A has an FK to node B, then Pebbles adds the edge $A \rightarrow B$ (overriding any pre-existing bi-directional edge detected from data flow propagation). In this way, foreign keys are precise but limited in coverage because they require programmers to specify them explicitly.

Increasing recall: Pebbles relies on one final relationship detection mechanism, *access relationships*. Access relationships can be seen as similar to data relationships, but while data relationships identify relationships as they are written to storage, access relationships identify relationships as they are read. Consider the case where an application has some data in memory that has not been synced to stable storage (and therefore is not yet tainted with any node's GUID). The app uses the data to generate the index for key-value object A and also writes that data into database row B. In the absence of explicit relationship information, we would hope that data propagation would detect the relation; however, it cannot because there is no data flow relationship when the data is written. We call this situation a *parallel write*, and resolve it by detecting data flow relationships when data is read

Property 4.1. Apps define explicit relationships through FKs in DBs, XML hierarchies, or FS hierarchies

Property 4.2. The SQLite database is the hub of all persisted data storage and access

Object Graph Construction Algorithm:

1. Data propagation: If data from A is written to B, then $A \leftrightarrow B$
2. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$ using Prop 4.1
3. Access propagation: If data from A is used to read B, then $A \leftrightarrow B$
4. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$, again using Prop 4.1
5. Utilize Prop 4.2, eliminating access based data propagation relationships that do not include any DB nodes.

Fig. 5: **Object Graph Construction Rules.**

in from storage: if data tainted with node A's GUID is used to access (read) node B, Pebbles adds $A \leftrightarrow B$ to the object graph. Again, this process is agnostic to the storage abstraction that the data is stored in, and relies only on data flow within the app. Access relationships can become an even greater source of imprecision than data relationships. For example, one could use data from one row, such as a timestamp, to select all the rows with that timestamp. Does that imply that all those rows should be considered as one object? Probably not.

Graph Generation Algorithm: Fig. 5 defines the algorithm used to construct the object graph, based on the observation that the DB is the hub of all persisted data. Step (1) leverages data flow propagation to construct a base graph, while (2) refines that graph by applying explicit relationship information. Step (3) applies access based data flow propagation to increase recall, and (4) again refines that graph with explicit relationship information. §7.1 evaluates LDO construction accuracy and precision in detail.

4.3 LDO Construction and Semantics

After constructing the object graph using the above semantics, Pebbles extracts the LDOs. Within the graph, an LDO is defined as the set of reachable nodes starting with a given node (the root of the object). Consider the email graph (Fig. 4), one can define a number of LDOs: an `Account` LDO, rooted in one `Account`-table row and containing multiple instances of five other row types, two files, and one XML entry; an `Email` LDO, rooted in one `Message`-table row and containing another row and one file, and so on. Although one LDO of each type is defined in the figure, in reality, there would be as many LDOs as there are instances of that type.

It is possible and correct for a single node to be part of multiple otherwise separate LDOs, in which case we say that the LDOs *overlap*. Consider, for instance, stateful accumulators (e.g. counts or sums over objects, stored in other objects), common resources (e.g. cache files that contain information about multiple objects), or log files.

Interface	Returned Objects
`getLDOContent(GUID, relevantOnly)`	LDO rooted at `GUID`
`getParentLDOs(GUID, relevantOnly)`	LDOs that contain `GUID`

Table 1: **The Pebbles API for Accessing LDOs.**

Pebbles exposes LDOs to protection tools via the Pebbles API, which consists of two functions (Table 1). `getLDOContent` returns the LDO rooted at the given GUID and `getParentLDOs` returns the LDOs containing the given GUID. Protection tools may specify with each call if only LDOs that may be relevant to the end-user should be returned.

4.4 From User-Level Objects to LDOs

Both of these API methods require an "object of interest" as a parameter. Pebbles provides a framework for protection tools to allow users to directly select an object of interest (from the user interface), and then use that object for future API calls. In this approach, a user enables a "marking mode" from a device-wide menu item, and then touches the item that they are interested in. Through taint tracking, we can determine the internal GUID for the object that was selected, and return that GUID back to the protection tool. This feature makes designing user-centric protection tools very easy: the tool need not concern itself with determining which objects to protect.

The mechanisms described thus far are useful for building a graph of all of an application's objects, but does not yet include a way to identify those objects that are relevant to users. For instance, in our email application there is another table, "sync_state," that stores how recently an account was synchronized with the server. Sync_state should clearly not be considered its own LDO, as its existence is essentially hidden from the end-user – the user will likely consider whatever data is stored here as, logically, part of the account. Pebbles leverages its system-wide taint tracking to identify which nodes in the object graph are directly displayed on the screen, Pebbles marks those objects (and other LDOs of the same type) as *relevant*. If an object is not relevant, then Pebbles will not allow it to be the root node of an LDO, instead including it as a member of the nearest parent node displayed on the screen.

5 Pebbles-based Tools

To showcase the value of Pebbles, we built four different applications that leverage its object graph.

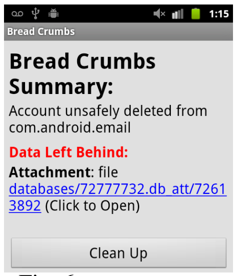
5.1 Breadcrumbs: Auditing Object Deletion

Motivated by Scenario 1 in §2.1, Breadcrumbs lets users audit the deletion of their objects – such as emails

Algorithm 1 Breadcrumbs Pseudocode

```
function WASFULLYDELETED(LDO $l$) $B \rightarrow$
    for all getLDOContent($l$) as $x$ do
        if $x$ exists still then Add $x \rightarrow B$
        end if
    end for
    for all $B$ as $x$ do
        Display $x$ and getParentLDOs($x$) to the user
    end for
end function
```

or documents – by their applications. It uses Pebbles's primitives to track objects as they are being deleted and identify any breadcrumbs left behind by the application.


Bread Crumbs

Bread Crumbs Summary:

Account unsafely deleted from com.android.email

Data Left Behind:

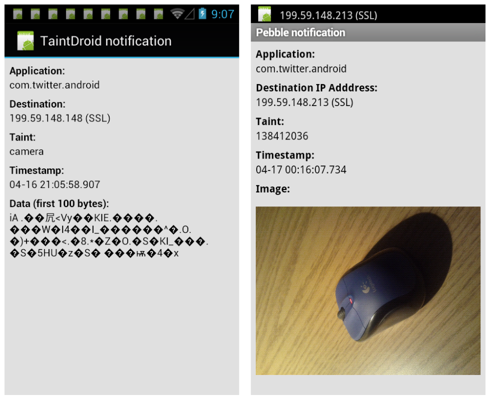
Attachment: file databases/72777732.db_att/72613892 (Click to Open)

Fig. 6: **Breadcrumbs.**

Users mark objects to audit for deletion (using Pebbles's object marking functionality), and then delete the object through their unmodified applications. They then open the Breadcrumbs application, which shows any persisted data related to recently tracked objects. In this way, users are not inundated with notifications about deletions and instead are only being presented with auditing information upon request. Fig. 6 shows a screenshot of Breadcrumbs's output when the user deletes an email in the Android email application. It shows the attachment file left behind and provides meaningful information about the leakage. A brief predefined interval after the user deletes a tracked object, Breadcrumbs destroys all relevant auditing information to protect the confidentiality of the partially deleted object.

Algorithm 1 shows how Breadcrumbs uses Pebbles's APIs to obtain all information necessary to identify and provide meaningful information about data left behind. Given a selected UI object, Pebbles identifies the GUID of the LDO represented by that LDO (as described in the previous section), and then Breadcrumbs calls `getLDOContent` to get all of its parts. For any part that still exists in persistent storage – the attachment file in this case – it displays meaningful metadata about that node. For example, instead of just showing the file's path, which can be nondescript, Breadcrumbs uses Pebbles's `getParentLDOs` function to retrieve the parent node, presumably a row. It displays the row's table name ("*Attachment*" in Fig.6), providing more context for information left behind. While the specific user interface we chose for Breadcrumbs can be improved, this example underscores the great value protection tools like Breadcrumbs can draw from understanding application-level object structures.

Our evaluation of Breadcrumbs on 50 apps (§7.3), reveals that incomplete deletions are surprisingly common:


9:07

TaintDroid notification

Application:
com.twitter.android

Destination:
199.59.148.148 (SSL)

Taint:
camera

Timestamp:
04-16 21:05:58.907

Data (first 100 bytes):
iA .��尻<Vy��KIE.����.
���W�I4��I_�������^�.O.
�)+���<.�8.*�Z�O.�S�KI_���.
�S�5HU�z�S� ���琉�4�x

199.59.148.213 (SSL)

Pebble notification

Application:
com.twitter.android

Destination IP Adddress:
199.59.148.213 (SSL)

Taint:
138412036

Timestamp:
04-17 00:16:07.734

Image:

Fig. 7: **Alert Screenshots.** (L): TaintDroid, (R): PebbleNotify.

18/50 apps leave breadcrumbs or refuse to delete objects from the local device.

Breadcrumbs could also be a useful tool for developers. A developer could proactively use Breadcrumbs to ensure that they are responsibly handling their user's data.

5.2 PebbleNotify: Tracking Object Exfiltration

Inspired by TaintDroid's data exfiltration tool [11], we built PebbleNotify, a tool that tracks exfiltration at a more meaningful object level. TaintDroid reveals data exfiltration at a coarse granularity: it can only tell a user that *some* data from *some* provider was exfiltrated from the device, but not the specific data that was leaked. For instance, consider a cloud-based photo editing application. A user might expect this application to upload the photo being edited to a server for processing; however, he may be interested in checking that no other photos are exfiltrated. Shown in the left hand side of Fig.7, TaintDroid would warn the user that data related to *some* photo was uploaded, but not *which* photo or *how many* photos. PebbleNotify is a 500 line of code application built atop Pebbles that interposes on the same taint sinks as TaintDroid, but provides object-level warnings. §6 describes in somewhat greater detail the modifications that we made to TaintDroid to track individual objects with high precision. Shown in the right hand side of Fig.7, it leverages application-level data structures exposed by Pebbles to give users meaningful, fine-grained information about their leaked objects.

5.3 PebbleDIFC: Object Level Access Control

As a logical extension to PebbleNotify, consider the case where rather than monitor the exfiltration of sensitive data, users want to prevent specific apps from having access to it. For example, in our previous example of a user using a cloud-based photo editing application, perhaps the user would rather simply prevent that photo editing app from having any access whatsoever to sensitive photos. PebbleDIFC supports this use-case by interposing on Android content providers, the mechanism used to share data between apps.

PebbleDIFC allows users to select individual objects that are sensitive, and then prevent them from being shared with other applications (in this case, photos). As with the rest of our protection tools, PebbleDIFC's implementation is straightforward. Before returning an object from a content provider, PebbleDIFC checks a table that maps apps to hidden objects, and prevents access to hidden objects.

5.4 *HideIt*: Object Level Hiding

Whereas PebbleDIFC allows objects to be permanently hidden from specific apps, *HideIt* supports a slightly different use case: allowing objects to be selectively hidden from all apps on the device, and then redisplayed at some later point, and perhaps hidden again later on. When objects are hidden (again, using Pebbles's marking mode), they are encrypted, and any record of their existence is filtered, by interposing on storage APIs. When objects are un-hidden, they are decrypted, and no longer filtered from API results. *HideIt* is intended for use-cases where small amounts of data need to be infrequently hidden from prying eyes, for instance, a parent lending their phone to their child.

5.5 Other Pebbles-based Tools

Although we designed and implemented Pebbles for Android, we believe that its object recognition mechanisms are applicable to other environments where a database is used as the hub of storage. In particular, we can imagine applying Pebbles as a software engineering tool to help developers understand either current or legacy applications where the database is the storage hub. A developer could use Pebbles to explore undocumented systems that do not make use of modern abstractions such as object relational mappers that would make the system easy to understand or to determine whether an application conforms to best practices and alert the developer if not. Understanding data structure from below the application could also enable testing tools and policy compliance auditing tools for cloud services [36]. We leave investigation of such applications for future work.

6 Implementation

We implemented Pebbles and each of the four above protection tools on Android 2.3.4 and TaintDroid 2.3.4. For Pebbles, we modify the SQLite, XML key/value store (a.k.a. SharedPreferences), and Java file system API to extract explicit structure, to intercept read/write/delete operations, and to register relationships. We also make several key changes to the TaintDroid tracking system, which we release as open source (`https://systems.cs.columbia.edu/projects/os-abstractions`). We next review our TaintDroid changes, after which we describe some implementation-level details of object graph creation.

TaintDroid Changes. To support Pebbles, we made three modifications to TaintDroid: (1) we increase the number of supported taints from 32 to several million, (2) we implement multi-tainting to allow objects to have an arbitrary number of taints simultaneously, and (3) we implement fine-grained tainting. The first two TaintDroid changes are necessary to track every row, file, and XML element with a separate taint and are implemented with a technique recently proposed in the context of another taint tracking system [26]. We omit the details here for space reasons.

The third TaintDroid change is motivated by massive taint explosion that we observed due to TaintDroid's coarse-grained tracking. Specifically, TaintDroid stores a single taint tag per String and Array [11]. Deemed a performance benefit in the paper, this coarse-grained tracking is unusable in Pebbles: we observed extremely imprecise object recognition and application-wide LDOs due to this poor granularity. As one example, CWMoney, a personal finance application, has an internal array that holds selection arguments used in database queries. This causes all nodes selected by that query to be related, defeating any hopes of object precision.

To address this problem, we modify TaintDroid to add fine-grained tainting of individual Array and String elements. To implement fine-grained tainting we add a shadow buffer to the Dalvik ArrayObject that contains the taint of each element in the array. If implemented naively, the shadow arrays would likely double the memory required for each array. To minimize the memory overhead from the shadow arrays we allocate the shadow array only when a tainted element is inserted into the array. This same optimization is implemented in [8]. Intuitively, only a small fraction of arrays in an device's memory should contain tainted elements (3-5% according to our evaluation). §7.2 shows that this lazy shadow array allocation significantly reduces the memory overhead of precise fine-grained tainting. We release our changes open source as a patch for TaintDroid.

Object Graph Implementation. The Pebbles graph is populated incrementally during application execution and persisted in a central database on the data partition so the graph does not need to be regenerated on each reboot. Applications interact with the Pebbles API through the Pebbles Object Manager that runs as part of the central system server process. Graph edges are generated on read and write operations to SQLite, shared preferences, and the file system. On read and write operations that generate new edges, requests for edge registration are placed on a queue within the application's memory space. This lets Pebbles perform bulk asynchronous registrations off of the main application thread improving application interactivity even during periods of heavy edge creation. In its current implementation the registration queue is not persisted to stable storage so it will be lost on application crashes or restarts. This is a potential attack vector that does not fall under the threat model for non-malicious applications.

7 Evaluation

We evaluate Pebbles over 50 popular applications downloaded from Google's Android market on a Nexus S running our modified version of Android 2.3.4. We seek answers three key questions:

Q1 *How accurate and precise is object identification in Pebbles?*

Q2 *What performance overhead does it introduce?*

Q3 *How useful are Pebbles and the tools running atop?*

Application Workloads. We chose 50 test applications from the top free apps within 10 different Google Play Store categories, including Books and Reference, Finance, and Productivity. We looked at the top 30 most popular applications within each category (by number of installs) and selected those that used stable storage. We also added a few open-source applications (e.g., OINote). The resulting list included: Email (Android's default email app), OINote (open-source note app), Browser (Android's default), CWMoney (personal finance app), Bloomberg (stocks app), and PodcastAddict (podcast app). For each application, our workload involved exercising it in natural ways according to manual scripts. For example, in Wunderlist, a todo list app, we created multiple lists, added items to each list, and browsed through its functions.

7.1 Pebbles Precision and Recall (Q1)

We measure the precision and recall of our object recognition by identifying how closely LDOs match real, application-level objects as users perceive them. We manually identified 68 potentially interesting LDO types across 50 popular applications (e.g., individual emails, folders, and accounts in the default email app; individual expenses, expense categories, and accounts in the CWMoney financial app). We evaluated whether Pebbles correctly identifies those objects (no leakage or over-inclusions). Recall measures the percentage of LDOs recognized without leakage; precision measures the percentage of LDOs recognized without over-inclusion.

To establish ground truth about LDO structure, we first populated the application with data and took a snapshot of the phone's disk, S_1, prior to creating the target object. Then, we created the object and took a second snapshot of the disk, S_2. The ground truth is the diff between S_2 and S_1 after manually excluding differences that are unrelated to the objects (e.g., timestamps in log files that differ between the two executions). We then exercised the application as thoroughly as possible so as to capture any edges that Pebbles might detect. To measure accuracy, we compare Pebbles-recognized LDOs to the

		Pebbles		File Tainting Only	
Application	**LDO**	**Detected**	**Precise**	**Detected**	**Precise**
Email	Account	Y	Y	Y	**N**
	Mailbox	Y	Y	Y	**N**
	Email	Y	Y	Y	**N**
OINote	Note	Y	Y	Y	**N**
Browser	History Item	Y	Y	Y	**N**
	Bookmark	Y	Y	Y	**N**
CWMoney	Account	Y	Y	Y	**N**
	Category	Y	Y	Y	**N**
	Expense	Y	Y	Y	**N**
Bloomberg	Stock	**N**	Y	Y	**N**
	Chart	Y	Y	Y	**N**
Podcast	Podcast	Y	Y	Y	**N**
	Episode	**N**	Y	Y	**N**
50 Total	68 Total	62/68 (91%)	66/68 (97%)	68/68 (100%)	0/68 (0%)

Table 2: **LDO Precision and Recall.** Sample applications and objects tested for object recognition precision and recall. "Y" indicates that an LDO was identified without leakage (column "Detected") or without over inclusion (column "Precise"). If an LDO has "Y" in both columns, its recognition is deemed correct. As expected, Pebbles performs far better than a straw man approach of treating entire files as a single LDO.

ground truth; if identical, we declare accurate recognition for that application and object.

Table 2 shows whether Pebbles correctly and precisely detects these LDOs. For comparison, we also evaluated the precision and recall of a basic approach, which represents perhaps the current state of the art: detecting relationships between files using just taint tracking and not using additional file structure to refine the granularity of objects. Pebbles correctly identifies 60 of the 68 objects across these 50 apps, without requiring any program modifications. Of the eight incorrectly identified objects, six were not correctly detected and two were not precise.

In each case that Pebbles failed to properly detect all components of the object (i.e., where it failed in recall), the leakage was due to a non-standard database specification. For instance, in the case of the app "ColorfulBudget", users can group expenses into categories, but Pebbles did not always properly detect the relationship between an expense and its category. Best practices would dictate that in such a case, all categories would be listed in a single table with a primary key (PK), and then each expense would contain a foreign key (FK) to reference the category's PK [4]. Traditionally this PK is an integer, to significantly increase lookup speed and decrease the amount of space needed to store any references to it [4]. However, in its current implementation, this app uses the actual name of the category as a key into the category table, without declaring such a dependency. Therefore, if a new category is created simultaneously with the creation of a new expense, we will experience a parallel write:

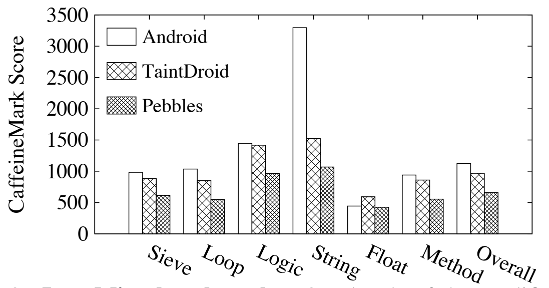


Fig. 8: **Java Microbenchmarks.** Overheads of the modified TaintDroid on the Java runtime with CaffeineMark, a standard Java benchmark. Higher values are better. Overheads on top of TaintDroid are 28-35%.

there will be no data dependence when the category is inserted and when the expense is inserted, since the category did not yet exist in storage. Moreover, since the relationship is not declared in the app schema as an FK, explicit relationship mechanism will not detect it.

While our access-based technique will largely eliminate this problem, there is still a gap when data is written but never read back. In these scenarios, such relationships could never be detected. Had these apps explicitly declared their DB relationships (e.g., in the above case by referencing each category by its PK), Pebbles would accurately recognized the objects.

As an example of Pebbles failing in precision (i.e., including additional objects as part of an LDO), consider the "Evernote" note taking app. Each time a notebook is updated, text in a SharedPreferences node is updated to reflect the newest notebook, creating a data dependency between the SharedPreference and the notebook. In this way, each notebook can become related to each other because Pebbles currently does not break data dependencies when text is updated. The only way that relations are broken in Pebbles is if an explicit relationship exists and is removed.

Without requiring any modifications to applications, Pebbles is able to achieve up to 91% recall or 97% precision. The straw man approach of utilizing only taint tracking (without knowledge of file structure) showed perfect recall (100%), and a complete failure in precision (0%). In other words, there were *no cases* of a single logical object stored in a single file. Overall, our results confirm that an unsupervised approach to application-level object recognition from within the OS works well, especially if schemas are relatively well-defined.

7.2 Performance Evaluation (Q2)

To evaluate Pebbles performance overheads, we ran two types of benchmarks: (1) *microbenchmarks*, which let us stress various components of our system, such as the computation and SQLite plugins; and (2) *macrobenchmarks*, which let us quantify our system's performance impact on user-visible application latency. Pebbles is built atop the taint tracking system TaintDroid

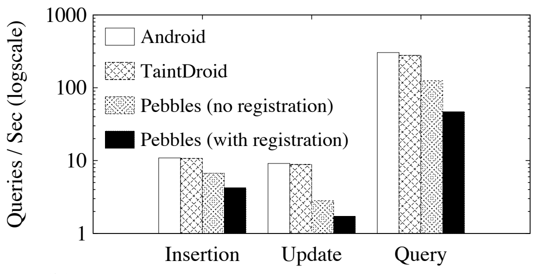


Fig. 9: **SQLite Microbenchmarks.** Overheads for various queries without and with relationship registrations.

[11], with several modifications made to increase taint precision (as discussed in §6). Therefore, we evaluate the performance overhead of Pebbles in comparison to both TaintDroid and to a stock Android device.

Microbenchmarks. Our first experiments evaluate the overhead of Pebbles with the Java benchmark CaffeineMark 3.0 [27] and are shown in Fig. 8. We ran the six computational benchmarks and find that Pebbles decreases the score by 32% compared to TaintDroid, which itself decreases the score by 16% compared to Android. The majority of this overhead comes from modifications to support more than 32 taints in Pebbles: TaintDroid combines tags by bitwise OR'ing, but Pebbles supports 2^{32} distinct taint markings, which are maintained in a lookup table. Pebbles also stores taint tags per individual array element, whereas TaintDroid stores only one taint tag per array, creating an additional overhead for Pebbles array-heavy benchmarks.

Pebbles also incorporates modifications to SQLite to detect and register relationships between rows with the Pebbles service. To evaluate the overhead, we compared the latency of simple, constant-size `SELECT`, `INSERT`, and `UPDATE` queries on an Pebbles-enabled Android versus Android. Fig. 9 shows query overheads when the query involves a relationship registration (59-168%) and when it does not (158-553%). No-registration queries – the cheapest to Pebbles – will likely be the common case for read-mostly workloads. For example, a document may be read many times, but relationship registration occurs only once. Moreover, batching and asynchronous-registration optimizations will likely help alleviate the overheads. The XML-based key/value store exhibits similar behavior, although we suppress concrete results.

Application-Level Performance. The above workloads are micro-benchmarks that stress the various components but do not necessarily relate to user-perceived performance impacts. To measure the impact of Pebbles on user-perceived interactivity, we evaluated the runtimes for various operations with three popular applications: Email, Browser and OINote. For Email, we look at app launch times and email reads; for Browser, we load the simple IANA homepage and the rich CNN and Google News pages over a local network; and for OINote we

App	Activity	Base	TDroid	Pebbles	Overhead
Email	Launch	196.8	202.1	260.0	63.2 ±1.11
	Load Email	211.6	253.6	463.6	252.0 ±1.64
OINote	Launch	182.6	229.4	219.7	37.2 ±1.58
	Load Note	59.5	70.2	84.9	25.4 ±0.14
Browser	Launch	96.5	124.0	148.1	51.6 ±1.63
	Load (iana)	154.0	209.3	395.3	241.4 ±2.26
	Load (CNN)	778.9	862.7	1443.1	664.2 ±17.56
	Load (GNews)	951.3	1023.5	1311.2	359.9 ±10.75

Table 3: **Application Performance.** Operation runtimes and overheads in milliseconds. 95% confidence interval shown for overhead. Base is the Android baseline, TDroid is TaintDroid.

read a note. All network access occurred over USB tethering to a host running a caching proxy; timing information excludes cache warmup. Table 3 shows the results in milliseconds. In almost all of the cases, overhead was less than 250ms. We saw more overhead and variation when rendering multimedia heavy web pages.

Memory Overheads. The modifications to TaintDroid to add fine grained tainting adds a memory overhead to the running system. We measure system wide memory usage while exercising three applications (Email, OINote, and Browser) with a similar workload as above. Without lazy memory allocation of array taint vectors (see §6), Pebbles's system-wide memory overheads are high: 188MB, 70MB, and 119MB, respectively, compared to TaintDroid. With lazy memory allocation, Pebbles exhibits much lower system-wide overheads: 34MB, 16MB, and 29MB, respectively. Although still higher than TaintDroid's own overhead of around 7MB for these applications, we believe Pebbles overheads are acceptable given devices' increased memory trends.

7.3 Case Study Evaluation (Q3)

Breadcrumbs. Using our Breadcrumbs prototype we evaluated deletion practices of 68 types of LDOs across 50 applications. Of the 50 applications, 18 of them exhibited some type of deletion malpractice.

Table 4 shows sample deletion malpractice. There were several cases where data from one LDO was written into another another and not cleaned up later. There were also several applications that did not delete items at the users' request, instead simply removing them from the user interface. We observed this in applications that heavily rely on cloud storage such as Wunderlist, a popular cloud-backed todo list application.

PebbleNotify. To evaluate PebbleNotify, we compared its output to that of TaintDroid Notify. When TaintDroid Notify detects that data tainted with a value from one of the selected sources is exfiltrated, it notifies the user with the application that is responsible for the network connection, the destination, the data source, the timestamp, and the first 100 bytes of the packet. This is useful metadata but it won't help a user learn specific information about the data being exfiltrated such as which picture or specific contact is leaving the device. We found that PebbleNotify was more informative because it shows a summary of the data being exfiltrated, and not just the metadata presented by TaintDroid Notify. PebbleNotify was particularly useful in the case of image exfiltration because it displays a thumbnail of the image being sent.

Application	Object Deletion Leakage
Email	Attachments remain after email/account deletion
ExpenseManager	Expenses remain after associated category deleted
Evernote	Notes/notebooks remain in database after deletion
On Track	Measurements remain after deleting category
14 other apps	21 LDO types unsafely deleted

Table 4: **Breadcrumbs Findings.** Shows samples of unsafe deletion in various applications.

PebbleDIFC. We integrated PebbleDIFC with the Android Media Provider and evaluated it by using it to mark several photographs on our device as sensitive (i.e., to prevent them from being shared). We then verified that those photos were not visible to applications other than the default Gallery application. We found that for this use case, PebbleDIFC has perfect accuracy: every photo that was marked was hidden, and no additional photos were hidden.

HideIt. We evaluated *HideIt* against many applications and largely found it to be effective. In our evaluation, we interacted with the application, populated it with data, and then marked a subset of the data as private so the application no longer had access. Interestingly, in most cases apps behaved as hoped when individual data objects were hidden and then again returned. There were however several cases where apps crashed when they expected some data to still exist, but was removed. We are interested in performing further investigations of the applicability of *HideIt*.

7.4 Anecdotal User Experience

To gain experience with Pebbles, the primary author carried it on his Nexus S phone for about a week. He primarily used the Email, Browser, Gallery, Camera, and PodcastAddict apps. We report two anecdotal observations from this experience. First, applications exhibit noticeable overhead during periods of intense I/O, such as on initial launch or when applications populate or refresh local stores. During regular operation we observed overheads that are anecdotally similar to ones exhibited by running Android 4.1 (a 2012 OS) on our Nexus S (a 2010 device). Second, to check if object recognition remains accurate over time, we examined at the end of the week the structures of a sample of the objects in our applications (e.g., emails, folders, photos, browser histories, and podcasts). We saw no evidence that object recognition degraded over time due to taint explosions or other potential sources of imprecision for Pebbles. Objects grew naturally; email folders grew in size to include relevant new email objects and they remained accurate.

7.5 Summary

Overall, our results show that: Pebbles is quite accurate in constructing LDOs in an unsupervised manner (*Q1*), performance remains reasonable when doing so (*Q2*), and data management tools can benefit from Pebbles to provide useful, consumer-grade functions to the users (*Q3*). In our experience, Pebbles either consistently identifies objects of a particular type (e.g., all emails, all documents, etc.), or it does not. Whether it works depends largely upon the application's own adherence to some common practices (described in the next section). When Pebbles works for all object types of an application, Pebbles can provide the desired guarantees under our threat model. And even when Pebbles is incomplete, it can still support transparency applications, improving visibility into data (mis)management of applications. Our accuracy results show that Pebbles discovers all object types in 42 out of 50 applications correctly (no over-inclusions/leakages). We leave development of tools to identify whether an application matches the Pebbles assumptions for future work.

8 Discussion

Pebbles leverages the structure inherently present in the storage abstractions commonly used on Android to identify LDOs. More formally, Pebbles assumes the usage of the following best practices:

R1: *Declare database schemas in full:* Given that the database is becoming the central point of all storage in modern OSes, having a well-defined database schema is important and natural. 42/50 apps we have evaluated in §7.1 meet such requirements sufficiently for Pebbles to work perfectly for them.

R2: *Use the database to index data within other storage systems:* A common programming pattern is to create a parent object (e.g., a message) in the database, obtain an auto-generated primary key, and then write any children objects (such as message body, attachment files) using the PK as a link. 47/50 apps use this pattern. We strongly recommend it to any programmers who need to store data outside the DB.

R3: *Use standard storage libraries or implement Pebbles storage API:* To avoid precision lapses, we recommend that apps use standard storage abstractions. As §3 shows, most apps already adhere to this practice: most apps use exclusively OS-embedded abstractions.

Relative to our evaluation of 50 apps, 39/50 adhere with all three recommendations, and 50/50 adhere with at least one of them. Pebbles' performance could suffer for apps that do not follow any of these recommendations. However, we believe that each recommendation is sufficiently intuitive and rooted in best practices to not impose undue burden.

9 Related Work

Taint Tracking for Protection and Auditing. Taint tracking systems (such as [3, 6, 17, 24, 31, 46, 49]) implement a dynamic data flow analysis that has been applied to many different context such as privacy auditing [6, 11, 48], malware analysis [24], and more [3, 49]. TaintDroid [11] provides taint tracking of unmodified Android applications through a modified Dalvik VM, a system that Pebbles builds upon for its object graph construction. To our knowledge, Pebbles is the first system to use taint tracking to discover data semantics of objects and provide a higher level abstraction with which to reason about and enforce such security properties.

Several systems utilize taint tracking to provide fine grained data protection and auditing. In each of these cases, however, a burden lies on the application developers to add hooks to identify relevant data structures to protection tool developers – a burden that could be lifted by Pebbles. For instance, CleanOS aims to minimize data exposure on a mobile device by automatically encrypting its "sensitive data objects" (SDOs) when not under active use [39]. The LDO abstraction is perhaps to some extent inspired by the SDO; however, SDOs must be manually specified by application developers, whereas LDOs are automatically identified and registered by Pebbles. Pebbles could be used to automatically identify SDOs, without requiring developer interaction.

Distributed information flow control (DIFC) systems such as Laminar [31], Asbestos [43], and Resin [46] let developers associate data with labels, and then allow either developers or end-users to specify security policies that apply to different labels. Taint tracking is performed during application execution to ensure that labels are propagated to derived data. Pebbles could be used to eliminate the need to statically annotate data with labels in code, instead automatically applying labels to LDOs as users request them. PebbleDIFC demonstrates the feasibility and power of such a system.

Related to taint tracking, data provenance [22, 23, 35] is close in spirit to logical data objects. It tracks the lineage of data (e.g., the user or process that created it). It has been proposed to identify the original authors of online information, to facilitate reproduction of scientific experiments [35], detect and avoid faulty data propagation in clouds [23], and others. It has to our knowledge never been used as an OS protection abstraction.

Fine-Grained Protection in Operating Systems. Many systems have been proposed in the past to support fine-grained, flexible protection in operating systems. Some of the earliest OSes, such as Hydra [45] and Multics [32], provided immense protection flexibility to applications and users. Over time, OSes removed more and more flexibility, being considered too difficult for programmers. Our goal is to eliminate the programmer from the loop by having the OS identifying objects.

More recently, OS security extension systems, such as SELinux [34] and its Android version, SEAndroid [33], extend Linux's access control with flexible policies that determine which users and processes can access which resources, such as files, network interfaces, etc. Our work is complementary to these, being concerned with external attacks, such as thieves, shoulder surfing, or spying by a user with whom the device has been willfully shared. Our abstractions, might, however, apply to SEAndroid to replace its antiquated file abstraction.

Securing and Hiding Data. Many encryption systems exist, operating largely at one of two levels of abstraction: block level [1, 21, 42] and file level [14, 16]. A drawback to such encrypted file systems is that it forces users to consider data as individual files, while logically there may be multiple objects that the user is interested in in a single file. Pebbles allows protection tool developers to provide a far finer level of control (at the object level) than these existing systems (at the file level).

Some protection tools are already operating at a higher level of data abstraction. These applications, such as Vault-Hide [25] and KeepSafe Vault [19], allow users to hide specific types of data, including photos, contacts, and SMSes. However, they only plug into a handful of supported apps and cannot provide generic protection for all apps. Pebbles aims to effect a similar level of control, but without requiring specialized work by protection tool developers to support specific applications.

Inferring Structure in Semistructured Data. Discovering data relationships is a key aspect of our work. Other have worked on inferring data relationships in various context: foreign key relationships in databases to improve querying [30, 47] and file relationships in OSes to enhance file search [37]. However, Pebbles can also infer relations among files, as well as other higher-level storage abstractions within modern operating systems. To perform such broad relationship detection, Pebbles differs significantly from other relationship detection systems in that it also leverages taint tracking.

Cozzie et al. developed the Laika system [9] which uses Bayesian analysis to infer data structures from memory images. Pebbles differs from Laika in that it does not attempt to recover programmer defined data structures but to discover application-level data relationships from stable storage that would be recognizable and useful to an end user or developer.

10 Conclusions

We have described *logical data objects* (LDOs), a new fine-grained protection abstraction for persistent data designed specifically to enable the development of protection tools at a new granularity. We described our implementation of LDOs for Android with *Pebbles*, a system that automatically reverse engineers LDOs from application-level persisted data resources – such as emails, documents, or bank accounts. Pebbles leverages the structural semantics available in modern persistent storage systems, together with a number of mechanisms rooted in taint tracking, to construct and maintain an object graph that tracks these LDOs without introducing any new programming models or APIs.

We have evaluated Pebbles and four novel protection tools that use it, showing it to be accurate, and sufficiently efficient to be used in practice to identify and manage LDOs. We can envision many other useful applications of Pebbles, such as data scrubbing or malware analysis, and hope that LDOs will enable the development of these and other granular data protection systems.

11 Acknowledgements

We thank our shepherd, Landon Cox and the anonymous reviewers for their valuable feedback, and Emmett Witchel for his support and advice. This work was supported by DARPA Contract FA8650-11-C-7190; NSF grants CNS-1351089, CCF-1302269, CCF-1161079, CNS-0905246, and CNS-1228843; NIH U54 CA121852; R01 LM011028-01; and Google and Microsoft gifts.

References

[1] dm-crypt: Linux kernel device-mapper crypto target. `https://code.google.com/p/cryptsetup/wiki/DMCrypt`, 2013.

[2] Anand Basu. Facebook Apps Leak User Information. `http://www.reuters.com/article/2010/10/18/us-facebook-idUSTRE69H0QS20101018`, 2010.

[3] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[4] Michael Brackett. *Data Resource Design: Reality Beyond Illusion.* IT Pro. Technics Publications Llc, 2012.

[5] Monica Chew. Writing for the 98%, blog post. `http://monica-at-mozilla.blogspot.com/2013/02/writing-for-98.html`, 2013.

[6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2004.

[7] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2005.

[8] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *Proceedings of the USENIX Security Symposium (Sec)*, 2014.

[9] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[13] Google. Storage options — android developers. `http://developer.android.com/guide/topics/data/data-storage.html`.

[14] Valient Gough. encfs. `www.arg0.net/encfs`, 2010.

[15] GRSecurity. Homepage of pax. `http://pax.grsecurity.net/`.

[16] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the Linux Symposium*, 2005.

[17] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[18] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2011.

[19] KeepSafe. Hide pictures - KeepSafe Vault. `https://play.google.com/store/apps/details?id=com.kii.safe`.

[20] Mary Madden and Aaron Smith. Reputation management and social media: How people monitor their identity and search for others online. `http://www.pewinternet.org/~/media/Files/Reports/2010/PIP_Reputation_Management_with_topline.pdf`, 2010.

[21] Microsoft Corporation. Windows 7 BitLocker executive overview. `http://technet.microsoft.com/en-us/library/dd548341(WS.10).aspx`, 2009.

[22] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

[23] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[24] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[25] NQ Mobile Security. Vault-Hide SMS, Pics & Videos. `https://play.google.com/store/apps/details?id=com.netqin.ps`.

[26] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. CloudFence: Data flow tracking as a

cloud service. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses*, 2013.

[27] Pendragon Software Corporation. Caffeinemark 3.0. `http://www.benchmarkhq.ru/cm30/`.

[28] Radia Perlman. File system design with assured delete. In *Proceedings of the IEEE International Security in Storage Workshop (SISW)*, 2005.

[29] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the USENIX Security Symposium (Sec)*, 2012.

[30] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2009.

[31] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[32] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM (CACM)*, 1974.

[33] SEAndroid. SEforAndroid. `http://selinuxproject.org/page/SEAndroid`.

[34] SELinux. Selinux project wiki. `http://selinuxproject.org/page/Main_Page`.

[35] Margo Seltzer. Pass: Provenance-aware storage systems. `http://www.eecs.harvard.edu/syrah/pass/`.

[36] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[37] Craig A.N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.

[38] Symantec Corporation. PGP whole disk encryption. `http://www.symantec.com/whole-disk-encryption`, 2012.

[39] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Mobile OS abstractions for managing sensitive data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[40] Yang Tang, Patrick P.C. Lee, John C.S. Lui, and Radia Perlman. FADE: Secure overlay cloud storage with file assured deletion. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[41] The Chaos Computing Club (CCC). CCC breaks Apple TouchID. `http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid`, 2013.

[42] TrueCrypt Foundation. Truecrypt – free open-source on-the-fly encryption. `http://www.truecrypt.org/`, 2007.

[43] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)*, 2007.

[44] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.

[45] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM (CACM)*, 1974.

[46] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[47] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 2010.

[48] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings*

of the ACM Conference on Computer and Communications Security (CCS), 2013.

[49] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 2011.