CleanOS: Limiting Mobile Data Exposure with Idle Eviction

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, Nikhil Sarda
Columbia University

Abstract

Mobile-device theft and loss have reached gigantic proportions. Despite these threats, today's mobile devices are saturated with sensitive information due to operating systems that never securely erase data and applications that hoard it on the vulnerable device for performance or convenience. This paper presents CleanOS, a new Android-based operating system that manages sensitive data rigorously and maintains a clean environment at all times. To do so, CleanOS leverages a key property of today's mobile applications – the use of trusted, cloud-based services. Specifically, CleanOS identifies and tracks sensitive data in RAM and on stable storage, encrypts it with a key, and evicts that key to the cloud when the data is not in active use on the device. We call this process *idle eviction* of sensitive data. To implement CleanOS, we used the TaintDroid mobile taint-tracking system to identify sensitive data locations and instrumented Android's Dalvik interpreter to securely evict that data after a specified period of non-use. Our experimental results show that CleanOS limits sensitive-data exposure drastically while incurring acceptable overheads on mobile networks.

1 Introduction

Mobile technology is replacing desktops as the primary personal computing platform and is being used in increasingly sensitive contexts. For example, today's users rely on smartphones and tablets to access their personal and corporate email, prepare tax returns, and review customer documents [32]. Even the US military recently announced that it will equip soldiers with Android devices for accessing classified documents [28]. The draw to new mobile technology is justifiable: mobile devices offer convenient and constant connectivity, increase productivity, and provide access to feature-rich, cloud-based applications (a.k.a. "apps").

Despite these advantages, the transition to mobile devices raises serious and yet unresolved concerns, particularly with respect to data security in the event of device theft and loss. Unlike desktops, generally assumed to be physically secure, mobile devices are extremely prone to theft and loss. Statistics here are staggering: 49% of the New York City population has experienced mobile phone loss/theft [24], and the FCC recently declared mobile theft an "epidemic" in major US cities [13].

Though alarming, these statistics have yet to prompt mobile OSes to address the serious data-security threats posed by device theft or loss. Like their desktop precursors, such as Linux and Mac OS X, mobile OSes let sensitive data accumulate uncontrollably on the device. For example, the OS accumulates significant amounts of data in cleartext memory, and the file system retains deleted files by not purging their contents. Despite being backed by clouds, applications hoard sensitive data – such as emails, documents, and banking information – on the vulnerable device. Although encrypted file systems [26], encrypted RAM [34], and remote-wipeout systems [3, 21] help protect this data, they are imperfect stopgaps for OSes that were simply not designed with physical insecurity in mind. For example, a recent study shows that 57% of corporate users employ no locking mechanisms on their smartphones, rendering encryption useless [32].

This paper presents *CleanOS*, a new Android-based mobile operating system[1] designed to *manage sensitive data rigorously and maintain a clean environment at all times in anticipation of device theft*. The crucial insight in CleanOS is to leverage the tight integration of today's mobile applications with trusted cloud-based services in order to evict sensitive in-memory and on-disk data to those services whenever it is not needed on the device. CleanOS thus ensures that the minimal amount of sensitive data is exposed on the vulnerable device at any time.

CleanOS extends Android in two major ways. First, it introduces *sensitive data objects* (SDOs), a new abstraction that facilitates management of sensitive data on mobile devices. An SDO is a logical collection of Java objects, files, and database items that applications create and use to manage their sensitive data, such as emails, financial data, or documents. SDOs and their data "disappear" from the device unless they are frequently used by an application. For example, if an email app adds an email's content to an SDO, any "trace" of that content automatically disappears from RAM and stable storage unless the user is actively reading that email on an unlocked screen. Recovering the email requires interaction with the cloud.

Second, to evict idle SDOs, CleanOS modifies Android's Java interpreter (Dalvik) to introduce a new type of Java garbage collector (GC), called an *evict-idle GC* (eiGC). While a traditional GC deallocates only those objects guaranteed to never be used in the future (i.e., no pointers to them exist), eiGC eliminates objects that have not been used for a period of time *even if* they might be used again in the future (i.e., pointers to them still exist). To do so, eiGC walks through all Java objects in an idle SDO and encrypts their data-bearing fields, such as primitives and arrays of primitives, with a key that is escrowed in the cloud. Our modified Dalvik interpreter then faults when a bytecode instruction executes on an evicted

[1] We view the OS notion broadly in this paper to include both the traditional OS and the entire Android framework on which apps run.

Component	New or Changed Features
Dalvik (JVM)	Evict-idle Garbage Collector (eiGC) Eviction-aware bytecode interpretation Secure deallocation of interpreted stacks
Android SDK	SDO API Default SDO heuristics
TaintDroid	Support for millions of taints SQLite vulnerability fix
SQLite	Taint tracking in database
Webkit	Screen-buffer purging
Bionic (libc)	Secure user-space deallocation
Linux Kernel	Secure page deallocation with grsecurity

Figure 1: **CleanOS Modifications to Android, TaintDroid.**

object, retrieves the key from the cloud, and decrypts the object. Thus, data eviction in CleanOS is logical; the data itself remains on the device in encrypted form, while the key is shipped to the cloud.

The major security benefit of CleanOS stems from the value-added services that app clouds can build on top of it. For example, a cloud could revoke data access following a theft report, provide an audit log of data exposed upon theft, or monitor data access to detect anomalous uses. Building such services on today's "dirty" devices would be tremendously challenging and likely require sacrificing semantics or performance. For example, Gmail allows email access revocation [18], but emails cached on the device remain exposed. Conversely, not caching sensitive data on the device degrades performance over slow mobile networks. CleanOS provides device-side OS support for building robust, secure, and efficient value-added cloud services.

We built CleanOS in Android using the TaintDroid taint-tracking system [12] and also implemented a value-added cloud service that provides post-theft data-exposure auditing. To do so, we modified several core components in Android and TaintDroid, summarized in Figure 1. Together, our changes provide: (1) eviction of idle Java objects, (2) heuristics for identifying sensitive data without requiring app changes, (3) support for millions of taints in TaintDroid, and (4) multi-layer secure deallocation of freed data in Java, native, and kernel space. While CleanOS' design extends in-memory eviction to stable storage, this paper and our current prototype focus on in-memory data eviction.

Overall, we make the following contributions:

1. We demonstrate the sensitive data exposure problem by analyzing 14 popular Android apps (§2).
2. We define SDOs, a new abstraction for managing sensitive data on theft-prone devices (§3).
3. We implement CleanOS, an Android OS extension that combines known encryption-based data destruction [4, 16, 30] with a new GC process that evicts idle sensitive data (§4 and §5).
4. We present a set of valuable add-on services that clouds could build on top of CleanOS (§6).

2 Case Study: Data Exposure on Android

We selected for analysis 14 Android apps according to their popularity in five sensitive categories: email, finance, document editing, password management, and social networking. We define as *exposed* any data that persists on the device – either in RAM or on storage – for a prolonged period of time, such as 10 minutes (§3 describes our rationale for this threat model). Our goal in the analysis was to answer three questions: (1) Is sensitive-data exposure a real problem? (2) If so, what are its causes? and (3) Is the exposure necessary? We tackle each question using examples from our analysis.

Is Data Exposure a Real Problem? We installed the 14 apps on a rooted Nexus S phone with Android 2.3.4 and asked the following question: what kinds of sensitive data can one find by dumping RAM and database contents while apps run in the background? Our acquisition process was vastly simplified by our rooted phone and the lack of encryption on the default Android configuration. Nevertheless, we believe that our findings indicate the level of data exposure on better-protected phones in face of realistic, albeit sophisticated, attacks, such as cold boot RAM imaging [19]. We created a stable-state environment – akin to the one a thief might find on a lost device – by ensuring that apps had not been used for 10 minutes prior to taking RAM and DB dumps.

The answer to our question is eye-opening: with simple techniques, we retrieved cleartext copies of sensitive information from *all but one app*. Figure 2(a) shows examples of cleartext sensitive data we extracted from a select subset of the apps. Figure 2(b), column *"Extracted Cleartext Data,"* expands the result set to all 14 apps and categorizes data in three classes of varied sensitivity: passwords, contents (e.g., email body, document content, bank account), and metadata (e.g., email subject, document title). Overall, we captured passwords in 5/14 apps, contents in 11/14 apps, and metadata in 13/14 apps.

What Causes Data Exposure? Given these results, an obvious question is what leads to so much leakage. There are several possible answers:

Insecure Deletion: The Android OS, including the kernel, system libraries, and the Java framework, leaks sensitive information by not erasing data securely after it is deallocated or by not securely erasing files when an app asks it to do so. These problems are well known in desktop and server settings and have been addressed with secure deallocation [6] and assured deletion [30, 39], respectively.

OS Data Buffering: Recent work shows that OSes and device drivers retain data in buffers past its intended life. It also shows how to limit OS-buffered data exposure [10].

App Data Hoarding: Although most of the apps are cloud-based, our experiments show that they hoard significant amounts of cleartext sensitive information on the device, either in RAM or in the local database. For exam-

App	Extracted Cleartext Data
Email	password, email contents, subjects, from/to, contacts
OI Notepad (doc)	document and metadata
KeePass (password mgr)	app password, all stored passwords & descriptions
Pageonce (finance)	password, transactions, bank account information
Facebook (social)	wall posts and messages

(a) Examples of data extracted from RAM / DB.

App	Description	Extracted Cleartext Data			Cleartext Data Hoarding: RAM			Cleartext Data Hoarding: SQLite DB		
		Pass-word	Cont-ents	Meta-data	Pass-word	Cont-ents	Meta-data	Pass-word	Cont-ents	Meta-data
Email	email (default)	Y	Y	Y	Y		Y	Y	Y	Y
GMail	email		Y	Y					Y	Y
Y! Mail	email	Y	Y	Y					Y	Y
GDocs	documents			Y			Y		Y	Y
OI Notepad	documents		Y	Y		Y	Y			
DropBox	documents			Y			Y			Y
KeePass	password mgr	Y	Y	Y	Y	Y	Y			
Keeper	password mgr	Y	Y	Y	Y		Y			
Amazon	commerce									
Pageonce	finance	Y	Y	Y	Y	Y	Y			Y
Mint	finance		Y	Y		Y	Y		Y	Y
Google+	social		Y	Y					Y	Y
Facebook	social		Y	Y					Y	Y
LinkedIn	social		Y	Y			Y		Y	Y

(b) Exposure of cleartext sensitive data across all 14 apps.

App	Data	When App Uses Data
Email	password	user/automatic refresh
	subjects	on the email list screen
	contents	user opens the email
OI Notepad	note title	on the note list screen
	note body	user edits the note
KeePass	master password	app launches
	entry name	on the entry list screen
	entry password	user opens the entry

(c) Example usage of hoarded data by apps.

Figure 2: **Sensitive Data Exposure.** (a) Examples of captured sensitive data. (b) A 'Y' indicates that we obtained cleartext copies from RAM/DB. A blank cell does not mean that the data is not on the device, but just that we did not find it in cleartext; the data could exist in some encrypted form. (c) Examples of when hoarded sensitive data is being actually used by the apps.

ple, the default Android email app maintains the email account password in cleartext RAM *at all times*, while KeePass, a popular password manager, loads its entire password database into RAM at startup and keeps it there. Column *"Cleartext Data Hoarding"* in Figure 2(b) shows the *persistent*, app-intended cleartext data we found in RAM or DBs.[2] It demonstrates that the hoarding behavior is pervasive: all but one of the 14 apps permanently maintain at least one type of sensitive data either in RAM or in the database, while 6/14 apps permanently maintain their passwords or some sensitive content in RAM.

Memory Leaks: Beyond the scope of our experiments is the well-known ease of unwittingly introducing memory leaks into Android applications [2]. If small, these leaks may go undetected and expose sensitive information.

Is Data Exposure Necessary? Although apps hoard significant amounts of sensitive data on mobile devices, they tend to access this data fairly infrequently, suggesting that data is often exposed longer than it needs to be. By way of example, Figure 2(c) identifies situations where three of our most problematic apps use hoarded sensitive data. For example, the password in the default Android Email app, which we know is exposed in RAM at all times, is in fact used only during inbox refreshes (the default is every 15 minutes). Similarly, each email's content is exposed in SQLite at all times but accessed only when the user opens that particular email. While the frequency of these operations depends on the workload, intuitively it should be relatively rare, making prolonged exposure unnecessary.

Implications for Mobile OS Design. Secure deletion for storage, RAM, and OS buffers has been acknowledged as, and developed into, a primary OS function [6, 30, 10]; however, the management of app-driven data hoarding or leakage has thus far been considered an app's own responsibility. For example, faced with similar data-hoarding practices in desktop and server applications, Chow, et al. [6] conclude that "little can be done without modifying the application" and that "leaks are recognized as bugs by application programmers, so they are actively sought after and fixed." Unfortunately, relying on the app to manage sensitive data is problematic. Sensitive-data caching presents tradeoffs between security on one hand and performance, usability, and energy/bandwidth consumption on the other hand. Without solid abstractions, calibrating these tradeoffs is challenging. For example, should a document-editing app cache the documents locally for good performance over cellular networks (as recommended in some mobile app guidelines [14]), or should it not do so for security reasons (as recommended in other guidelines [40])? Should it cache the user's password for convenience, or should it prompt the user for it whenever it is needed?

We argue that mobile OSes *can* and *should* offer abstractions for apps to manage their sensitive data rigorously *without* sacrificing their performance, usability, or other properties. This paper introduces one such abstraction in CleanOS, whose goals we next describe.

3 Goals and Assumptions

Goals. The primary goal of CleanOS is to minimize the exposure of an app's *allocated* sensitive data by evicting it from the device whenever the data is idle (i.e., not being actively used by the application). The key insight that makes this possible is the tight integration between today's mobile apps and cloud services. CleanOS leverages clouds to create a new abstraction, called a *sensitive data object* (SDO). SDOs track sensitive information as

[2]For RAM, we conservatively assume an object to be persistent if it always appears in the app's Java object dump.

it flows through RAM and stable storage. As soon as they become idle, they are automatically evicted to the cloud and are recovered only when the app needs them again.

Specific design goals of CleanOS include:

1. *Eviction:* SDOs should "disappear" as soon as they become idle whether or not they are expected to be used by an application in the future.
2. *Reasonable performance:* We seek to provide reasonable performance for popular mobile apps despite data eviction over Wi-Fi or cellular networks.
3. *Reasonable defaults:* While we admit app changes for best performance and semantics, we aim to offer reasonable defaults even for unmodified apps.
4. *Leverage technology trends:* CleanOS must integrate naturally with existing tech trends, such as the tight integration of mobile apps with cloud services.
5. *Design for mobiles:* CleanOS' design should target mainstream mobile technologies, such as Android.

Eviction of idle data (Goal 1) is our primary goal and contribution in CleanOS. We strive to ensure that a thief cannot get a "free lunch" by capturing a device. Rather, he should be required to contact the cloud in order to access data of interest, at which time the cloud could deny access, log it, rate-limit it, etc. However, enforcement of precise timeouts on idle sensitive data is a non-goal. From a performance perspective (Goal 2), we wish to ensure that popular apps remain usable despite eviction across Wi-Fi or cellular networks (e.g., 3G/4G).

A common pitfall when proposing new OS abstractions is to require application changes to gain any benefit. To avoid this, CleanOS should include heuristics to construct default SDOs that provide reasonable eviction and performance properties even for unmodified apps (Goal 3). Finally, we aim to exploit unique properties of popular mobile technologies in CleanOS' design (Goals 4 and 5). First, we leverage the tight integration between most mobile apps and trusted cloud services to evict device data to those services. For local-only apps, however, the user can still integrate them with his own CleanOS service. Second, while the data eviction concept is applicable to any mobile OS, we focus our design on Android, which lets us leverage its technological properties to facilitate data eviction. For example, since all Android apps are written in Java, we decided to tap into the garbage collector to evict idle sensitive Java objects.

Threat Model. Our threat model considers *any data on a mobile device to be vulnerable to data-driven thieves*. While many data protection systems exist – including encrypted file systems [26, 38, 11], encrypted RAM [34, 23, 31], and data wipeout systems [3, 21] – they are imperfect when confronted with negligent users or (sophisticated) physical attacks. First, users can foil any protection system by not locking their devices [32], assigning trivial PINs or passwords [20], or writing passwords down in easily retrievable locations [36]. Second, mobile devices are prone to physical attacks, which are notoriously difficult to protect against. For example, an attacker could use cold boot attacks [19] to retrieve in-RAM decryption keys or data, break the seal of tamper-resistant hardware [1, 35], or shield the device from the network to prevent remote wipeout [3]. Such threats are especially relevant for corporate, government, and military users, who interact with particularly sensitive data, such as trade secrets, customer data, health data, or state secrets.

To maintain post-loss control over data despite such threats, CleanOS evicts data to a cloud service, which is assumed to be trusted and non-compromisable. In reality, mobile users are already required to trust the clouds on which their apps rely, so our assumption is reasonable. Depending on the deployment model, these clouds could integrate directly into CleanOS to help cleanse their apps automatically. For apps without a cloud component, we assume that users can evict data to a trusted community or self-administered CleanOS service. Finally, we assume that the cloud learns about a monitored device's theft, either directly from the user or via an automatic mobile-theft detection mechanism.

CleanOS explicitly assumes that the mobile device, along with all software running on it, is trusted until it is lost. For example, the thief cannot install malware on a user's device, tamper with the device physically, or inspect it prior to stealing the device. After loss, we trust neither the hardware nor the software on the device.

We assume that disconnection is the exception rather than the rule. With pervasive wireless and cellular network coverage, this assumption is becoming increasingly realistic. Moreover, CleanOS is especially geared toward cloud-based apps, which typically require connectivity for full functionality. Nevertheless, we present techniques to allow disconnected operation in certain cases.

CleanOS is most applicable to long-lived daemon-like apps, whose execution consists of brief computation sessions interspersed with long periods of inactivity. Most of today's mobile apps follow this model, including email, browsers, document editors, and social apps. CleanOS disables exposure during periods of inactivity.

Finally, we explicitly assume the existence of robust secure deallocation and OS buffer-cleanup techniques [6, 30, 10] and do not aim to improve the state of the art in these intensely-researched directions. Rather, we focus on limiting the exposure of sensitive data that *applications* hoard or leak, a problem previously thought intractable from an OS perspective (see §2).

4 The CleanOS Architecture

We now describe our CleanOS design for Android. We focus initially on in-RAM data eviction, after which we show how to extend SDOs to stable storage.

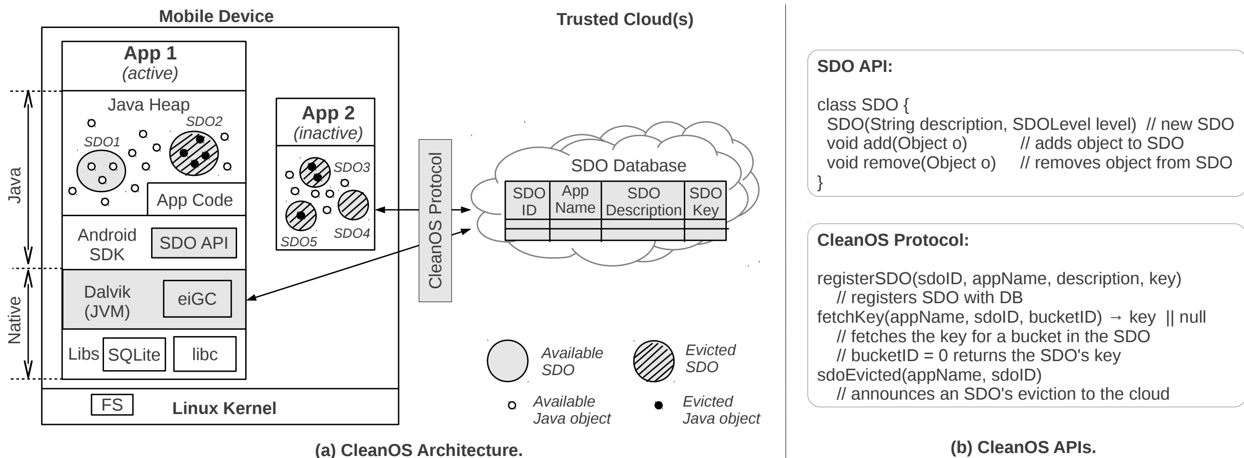


Figure 3: **The CleanOS Architecture and APIs.** (a) The architecture, with key components highlighted in grey. We add or modify in some way all of the boxed components (except for FS and kernel). (b) The CleanOS SDO API and device-cloud protocol.

4.1 CleanOS Overview

Figure 3(a) shows the CleanOS architecture, which includes three major components: (1) the *sensitive data object* (SDO) abstraction, (2) a modified, eviction-aware version of the Dalvik interpreter, along with an *evict-idle garbage collector* (eiGC), and (3) the SDO cloud store. Briefly, apps create SDOs and place their sensitive Java objects in them. The modified Dalvik tracks their propagation across RAM with TaintDroid and monitors their bytecode-level accesses. The eiGC evicts SDOs to the cloud if they remain idle for a specified period.

An SDO is a logical collection of Java objects, such as string objects representing the emails in a thread or objects pertaining to a bank account in a finance app. Upon creation, SDOs are assigned app-wide unique IDs and encryption keys (K_{SDO}), and are registered with the cloud.

We implement three functions for SDOs. First, we track objects in an SDO with a modified TaintDroid system, using the ID as a taint. As objects are tainted with an SDO's ID, they become part of the SDO. For example, `SDO1` in Figure 3(a) includes three objects added to it either explicitly by the app or automatically by our modified Android framework. Second, we monitor accesses to SDOs and record their timings. Whenever an app accesses an object in an SDO (e.g., to compute on it, send it over the network, or display it on screen), that SDO is marked as used. Third, we evict SDOs when they are idle for a time period (e.g., one minute).

To evict idle SDOs, the eiGC eliminates unused Java objects from RAM *even if* they are still reachable. It periodically sweeps through Java objects and evicts them if they are tainted with an idle SDO's ID. In Figure 3(a), the active app (App 1) has one available SDO (`SDO1`) and one evicted SDO (`SDO2`). For example, an SDO associated with an email thread might be available while the user reads emails in that thread, but the password SDO might remain evicted. When the app goes into the background, all of its SDOs might be evicted, as shown for App 2. An SDO is evicted when all Java objects in it have been evicted; however, an available SDO may have both evicted and available Java objects.

Conceptually, eviction occurs at the level of logical SDOs. In practice, however, CleanOS must eliminate the actual data-bearing objects from the vulnerable device. To do so, eiGC leverages encryption-based data destruction from assured-delete file systems [30, 16] and applies it to the memory subsystem. Specifically, eiGC replaces data-bearing fields in objects, such as primitives and arrays of primitives, with encrypted versions and then securely destroys the encryption key. To encrypt a data field F, eiGC uses a key K_F that is uniquely generated from the SDO's key K_{SDO} in the cloud (see §5 for details). We modified Dalvik to fault when an app attempts to access the evicted data, at which time it retrieves K_{SDO} from the cloud, generates K_F, and decrypts the data. K_{SDO} is then cached onto the device and securely removed when the SDO as a whole is again evicted.

We next provide more detail on the two main contributions of CleanOS: the SDO abstraction and the eiGC.

4.2 The SDO Abstraction

SDOs fulfill two functions in CleanOS. First, they let CleanOS identify sensitive data and focus its cleansing on that data for improved performance. Indeed, evicting all Java objects indiscriminately would be prohibitively expensive, while evicting a few at random would diminish security benefits. Second, SDOs are instrumental in supporting some of our envisioned add-on cloud services, such as the auditing service described in §6, as they identify and classify sensitive data for the auditor.

APIs. Figure 3(b) shows the SDO API. To realize the data-control benefits of CleanOS, apps create SDOs and `add`/`remove` Java objects to/from them. To create an SDO, an app specifies a description, which is a short, human-readable string that describes the sensitive data

associated with the SDO. For example, our modified email app, CleanEmail, creates an SDO using "*password*" for the description and `adds` the password object to it. It also creates one SDO for each email thread, specifying the thread's subject as the description, and `adds` each email in the thread to it. Section 6 describes two apps that we trivially ported to CleanOS with minimal modifications (fewer than 10 LoC).

Figure 3(b) also shows the protocol used to register SDOs, retrieve their keys after eviction, and report their eviction to the cloud. To create an SDO, the app registers it with the cloud database using the SDO API, specifying its ID, the app's package name (such as `com.android.email`), the description, and the encryption key. For example, the description for an SDO associated with a certain thread might be the subject of that thread. In the database (whose schema is included in Figure 3(a)), the tuple ⟨app package name, SDO ID⟩ is a phone-wide unique identifier. Although not implemented in our current prototype, the database can use the app user id to restrict access to keys only to the apps that created them. Finally, to enable auditing services such as Keypad [15], CleanOS notifies the cloud asynchronously whenever it evicts an SDO (message `sdoEvicted`). The notification is needed because, unlike Keypad, CleanOS does not forcibly evict keys at an exact time after they were fetched; rather, it does so when convenient, depending on load and networking conditions (see §4.5 and §5).

Default SDOs. As mentioned in §3 (Goal 3), we aim not to rely on app modifications to gain tangible benefit from CleanOS. To this end, we modified the Android framework to register a set of default SDOs and use simple heuristics to identify and classify Java objects coarsely on behalf of the apps. For example, our current prototype creates several default SDOs by plugging into various core classes in the SDK: a *User Input* SDO for all input a user types into the keypad (class `InputConnection`), a *Password* SDO for any Java objects that capture input a user types into a password field (based on attributes of class `TextView`), a coarse *SSL* SDO for all objects read from incoming SSL connections (class `SSLSocket`), and SDOs for input from particularly sensitive sensors, such as the camera. Some of these heuristics (e.g., SSL) were inspired by prior work on automatic identification of sensitive data [9]. Although default SDOs are coarse and may potentially include many non-sensitive objects (particularly SSL), we believe that they offer comprehensive identification of most sensitive data in unmodified apps. For example, all the sensitive data we analyzed in §2 would be capturable by a default SDO. For apps willing to adapt, CleanOS allows the overriding of default assignments of objects to SDOs.

Eviction Granularities and Buckets. Thus far, eviction granularities have been determined by SDOs, which is problematic for two reasons. First, it forces app writers to consider granularities and taint propagation when they design their SDOs. Second, our default SDOs, such as SSL, are coarse. In our view, CleanOS should offer eviction benefits even when an app dumps all of its sensitive objects into one big SDO, e.g., "*sensitive*."

To support fine-grained eviction with coarse-grained SDOs, we introduce *buckets*. Specifically, an SDO is "split" into several disjoint buckets, which are evicted independently. Java objects added to the SDO – either by the app or by our framework – are placed in random buckets. Eviction occurs at the bucket level: when a bucket has been idle for a period, all objects in it will be evicted using a bucket key, which is derived from the SDO's key using a key derivation function [22]. For example, in an unmodified email app, we would place all emails into the *SSL* SDO. With buckets, different emails would be placed into different buckets of the *SSL* SDO and might therefore be evicted independently. Also, with bucketing, we cache bucket keys instead of SDO keys on the device.

4.3 The Evict-Idle Garbage Collector

A simple but important innovation in CleanOS is the evict-idle GC (eiGC). At its core (and independent of CleanOS), eiGC implements for a managed language what swaping implements for OSes: it monitors when objects are being accessed during bytecode interpretation and evicts them when they have not been used for a while. We believe that the eiGC concept has applications beyond CleanOS, such as limiting the amount of memory used by Java applications on memory-constrained devices at a finer grain than OS-level paging would be able to sustain. In the context of CleanOS, however, eiGC evicts Java objects in idle SDO buckets.

Using the GC to evict sensitive data is not the only design worth considering when building a "clean" OS. We contemplated modifying the kernel's paging mechanism to swap idle pages to a Keypad-like encrypted file system [15], which at its core achieves for files a similar eviction function to the one we achieve for RAM. We chose the GC approach for two reasons. First, evicting Java objects provides finer-grained control over sensitive-data lifetime than full-page eviction. Second, by evicting at the JVM level we can leverage TaintDroid, the only taint tracking system for Android. Tracking sensitive data is vital for constructing the SDO abstraction, which in turn is the base for building powerful add-on services, such as auditing. However, our decision has a downside: coverage. By evicting Java objects, we may miss data intentionally maintained by native libraries. We discuss this limitation further in §8.

4.4 SDO Extension to Stable Storage

Like RAM, stable storage requires sanitization. At first glance, systems such as Keypad [15] could be directly

leveraged to evict unused files in CleanOS. Unfortunately, we found that eviction at file granularity is unsuitable for Android, where apps typically rely on a database layer to manage their data. For example, 11 of the 14 apps in Figure 2(b) store their data in SQLite, which maps entire databases as single files in the FS. As a result, if the DB file were exposed, then *all* of its items would be exposed, including long-unaccessed emails and documents.

CleanOS tailors storage eviction specifically for Android by extending the in-RAM SDO abstraction to include files *and* individual database items. For this, we use two mechanisms. First, we propagate SDO taints to files and database items. Unfortunately, TaintDroid supports only the former, not the latter, an important vulnerability we discuss in §5. We fixed this in CleanOS by modifying the SQLite DB. Specifically, we automatically alter the schema of any table to include for each data column, *C*, a new column, *Taint_C*, which stores the taint for each item in that column (SDO ID and bucket ID). Second, before storing a tainted data object in a DB, we first *evict* that object, i.e., encrypt it with its eviction key. When the database needs the object, it must decrypt it.

4.5 Disconnected Operation

While we assume that disconnection is the exceptional case, we present techniques to deal with two types of disconnection: (1) short-term disconnection, such as temporary connectivity glitches, and (2) long-term, predictable disconnection, such as a disconnection during a flight. To address short-term disconnection, we can extend eviction of already available SDOs by a bounded amount of time (e.g., tens of minutes). This allows an app to continue executing normally while temporarily disconnected until it reaches an evicted object. For example, a user might be able to load recently accessed emails, but not older ones.

To address long-term disconnection, such as during air travel, we hoard SDO keys before entering into disconnection mode. For example, our prototype implements Dalvik support for hoarding SDO keys upon receipt of a signal. We plan to wrap this functionality into a privileged app that provides users with a “Prepare for Disconnection” button, which they can press before boarding a flight. To prevent a thief from using this button to retrieve all SDO keys, the cloud would require the user to enter a password. While we generally shun user-configured passwords in CleanOS, we believe that long-term disconnection is a sufficiently rare case to warrant enforcement of particularly strong password rules with limited impact on usability [27]. In contrast, imposing such rules on frequent unlock operations would be impractical.

4.6 Deployment Models

CleanOS presents multiple deployment opportunities. First, security-conscious apps can use their own, dedicated clouds to host keys and provide add-on services, such as auditing. In such cases, we expect that the mobile side of apps would define meaningful SDOs. Second, users who are particularly concerned with apps that have not yet integrated with CleanOS might use a CleanOS cloud offered by a third party or that they host themselves. For example, our prototype hosts all keys for all apps on a Google App Engine service that we implemented.

5 Prototype Implementation

We built a CleanOS prototype by modifying Android 2.3.4 and TaintDroid in significant ways (see Figure 1). To date, our prototype fully implements eviction of in-memory SDOs and propagates taints to SQLite, but it does not yet encrypt sensitive items in SQLite. Doing so will require changing the native part of the SQLite library – a single, massive, over-100K-LoC file – the major deterrant we encountered thus far. We next describe modifications we made to components of particular interest.

TaintDroid with Millions of Taints. Most dynamic taint-tracking systems, including TaintDroid, support limited numbers of taints, which would prevent CleanOS from scaling to many SDOs. For example, TaintDroid supports only 32 taints by representing them as 32-bit shadow tags, where each taint corresponds to one tag bit. This limitation allows propagation of multiple taints on one object for tracking completeness and security against malicious applications. For CleanOS, which trusts applications, we modified propagation to allow many taints.

We rely on a simple observation, which we validate experimentally: in practice, when multi-tainting occurs, we can usually define a strict, natural ranking for taints in terms of their sensitivity. As intuitive examples, a *Password* SDO should be more sensitive than a generic *User Input* SDO, and a KeePass secret’s SDO should be more sensitive than its description SDO. In these cases, “losing” the less sensitive taint would be admissible, because it does not weaken the user’s perception of the gravity of an object’s exposure. Using a 24-hour real-usage trace for the Email app (see §7.1), we confirmed that 98.8% of the tainted objects were either assigned a single taint during their lifetimes or received multiple taints whose sensitivity could be strictly ordered using a simple, static, three-level ranking system: `HIGH`, `MEDIUM`, and `LOW`. The remaining 1.2% of the objects received multiple taints of undecidable ordering within this ranking system (i.e., equal sensitivity levels). Similar traces for Facebook and Mint indicated even fewer undecidable cases ($< 0.01\%$).

Based on this observation, we introduce the concept of *sensitivity level* for taints and use it to propagate a single taint per object. Apps specify a sensitivity level for each SDO upon its creation. If an object were added to two SDOs during taint propagation, CleanOS retains the one with the higher sensitivity level. For equal sensitivities (the rare case), CleanOS retains the most recent

Figure 4: **CleanOS Taint Tag Structure.** We impose a structure on TaintDroid taints to support arbitrary numbers of taints.

taint. Figure 4 illustrates the revised structure for the taint tag, in which we pack together the sensitivity level, SDO ID, and bucket ID into 32 bits while supporting up to 2^{25} SDOs. In our experience, assigning sensitivity levels to SDOs is natural, as demonstrated in §6. The idea of propagating a single taint was used before in hardware-based taint tracking systems for improved performance [5].

Eviction-Aware Interpretation in Dalvik. We reserved the most significant bit in the taint tag to denote the eviction state of a field. We modified the Dex bytecode instructions that access object instance fields and array members. This includes instructions such as `OP_AGET`, `OP_IGET`, `OP_SGET` (used to retrieve array members, instance fields, and static fields, respectively). Our new instruction implementations first test the value of the eviction bit in the field's taint tag. When the bit is set, we request the aforementioned K_{SDO} and decrypt the value before allowing the instruction to proceed. If a key is not available, execution is suspended.

The Evict-Idle Garbage Collector. While eiGC walks the reachable objects, we inspect the taint tag for each object field and retrieve its idle time. If it exceeds the configured threshold, then eiGC retrieves the key associated with the tag and encrypts the value. Only fields that represent actual data are evicted (primitives and arrays of primitives); fields implemented as pointers are not evicted, as a pointer is not in and of itself sensitive.

To evict data, we use AES in counter mode to generate a keystream, which we use as input to an XOR operation with each byte of the data to be evicted. The size of the keystream depends on the data's type. For primitives, it is either 4 bytes (for `char`, `int`, `float`, etc.) or 8 bytes (for `double` or `long`). For arrays, many bytes may be necessary. We use the bucket key to generate an appropriately sized keystream. For primitives, we replace the data with a pointer to a structure containing metadata necessary for decryption (e.g., initialization vectors) and the resulting ciphertext. For arrays, we evict the contents in place and store the necessary metadata inside the ArrayObject.

Running the eiGC continuously would prevent the CPU from turning off when the mobile device is idle, thereby wasting energy. Fortunately, eiGC needs to run only while sensitive objects are left unevicted. Hence, in our prototype, eiGC stops executing as soon as it has evicted all data, which should occur shortly after the app goes idle. The eiGC resumes execution once the app faults on an evicted object or assigns a new taint to an object. Hence, eiGC runs only while the app also runs.

Optimizations: Bulk Eviction and Prefetching. Performance and energy are major concerns with CleanOS, for two reasons. First, garbage collection is expensive; hence performing it frequently hurts app performance and energy (e.g., the eiGC's full-heap scans block interpretation for 1-2s). Second, our reliance on the network to fetch decryption keys causes app delays and dissipates energy.

To address the first problem, we developed *bulk eviction*, in which the eiGC evicts sensitive Java objects *all at once*, soon after the app itself becomes idle. More specifically, while the app is executing, we evict nothing and perform no GC; once the app has remained idle for a predefined time (e.g., one minute), the eiGC performs a full-heap scan-through and evicts all cleartext tainted objects. This technique reduces the number of heavyweight GCs to just one per app execution session, thereby minimizing the eiGC's impact on performance and energy.

To address the second problem, we developed *bulk key prefetch*, which prefetches all keys that were accessed during the last eviction period upon the app's first miss on a key. For example, if a user opened his inbox subject list and read two emails during a previous interaction session with his email app, then the next time the user brings the app into the foreground, CleanOS will fetch the decryption keys for the subjects and the two emails' contents – all in one network request. If the user views only his subject list but reads no emails in a previous session, then the next time around, CleanOS will fetch only subject keys again, not any email content keys. This technique improves app launches and the latency of repeated operations, such as re-reading an email. It can be extended to prefetch keys used in the last *N* sessions.

Although these optimizations may improve performance and energy, they may also increase sensitive-data exposure. For example, prefetching previously-used keys may expose some sensitive data needlessly. We quantify this performance/exposure tradeoff in §7.2.

Multi-Level Secure Memory Deallocation. Android goes to great lengths to keep an application running in the background so it can re-launch quickly. This can cause an accumulation of sensitive data in areas of memory that are no longer in use but have not been returned to the kernel. The object heap in Dalvik is implemented using dlmalloc mspaces and relies on the implementation of `free()` in dlmalloc to return memory to the mspace. To implement secure deallocation, we changed both `free()` and an Android-specific modification to dlmalloc that merges chunks of adjacent free memory. These functions now overwrite the space being released with a fixed pattern. We also modified Dalvik to overwrite interpreted stack frames on method exit, scrubbing them of sensitive data. Finally, when assigning default taints to Java objects, we made explicit efforts to taint objects as soon as they enter Java space from native libraries.

Addressing a TaintDroid Vulnerability. When implementing CleanOS, we uncovered a surprising implication

of a known limitation in TaintDroid. Specifically, TaintDroid does not track changes in native libraries, which, as acknowledged by its authors, may allow a *malicious* library to leak tainted data without triggering an audit log. To address this problem, TaintDroid prevents untrusted apps from loading any native libraries other than system libraries (e.g., SQLite and WebKit), which are included in Android itself and are therefore *trusted*. This measure has thus far been thought sufficient.

Nevertheless, we discovered that even *trusted* system libraries can be exploited by a malicious app to expose tainted data with no alarms. For example, because SQLite is written in native code, a malicious app could wash taints off a tracked data item simply by storing it into the database and reading it back. More generally, any stateful libraries that provide the ability to put and later retrieve data are vulnerable to attacks. Since disabling system libraries is impractical (e.g., 12/14 apps in §2 depend on SQLite), we instead suggest identifying and modifying all stateful system libraries to propagate taints.

To date, we modified two such libraries: SQLite and WebKit. For SQLite, we implemented taint propagation by persisting taints along with the data (see §4.4). For WebKit, we disabled caching of rendered Web pages. While important for security, we leave identifying and fixing other libraries for future work and for now suggest notifying the cloud about a potential leak if sensitive data were handed over to an unchecked native library. We suggest that TaintDroid proceed similarly. We discuss the coverage limitation further in §8.

6 Applications

We ported three of our "dirtiest" apps from §2 onto CleanOS and built a proof-of-concept, add-on service.

6.1 Extending Apps with SDOs

Although unmodified apps can benefit from the coarse default SDOs that CleanOS offers, they can also define their own SDOs for fine-grained control of sensitive data. To demonstrate how apps can be "ported" to our API, we modified two open-source apps – Email and KeePass – to define fine-grained SDOs. Changes for both apps were trivial. For Email, we added these seven lines of code:

```
SDO subjectSDO = new SDO("Subject", SDO.LOW);
subjectSDO.add(mSubject);
SDO bodySDO = new SDO("Content␣of␣" + mSubject, SDO.MED);
bodySDO.add(mTextContent);
bodySDO.add(mHtmlContent);
bodySDO.add(mTextReply);
bodySDO.add(mHtmlReply);
```

We added each email's subject to a global, low-sensitivity SDO and created a medium-sensitivity content SDO for its body, using the subject itself as the description. Passwords, already embedded in an SDO by our default heuristics, needed no changes.

For KeePass, changes were similarly trivial (7 lines):

device	message	time
cd5493c1befeb9075442862afa046182	fetchKey(9.408 - com.android.email - password)	2012-04-28 17:26:50.590000
cd5493c1befeb9075442862afa046182	registerSDO(com.android.email - Invitation to develop "Clean OS")	2012-04-28 17:27:01.140000
cd5493c1befeb9075442862afa046182	fetchKey(30.709 - com.android.keepass - Entry)	2012-04-28 17:27:48.500000

Figure 5: **Screenshot of Audit Service Log in App Engine.**

```
SDO masterSDO = new SDO("Master␣key", SDO.MED);
SDO entrySDO = new SDO("Entry", SDO.HIGH);
masterSDO.add(mPassword); // In SetPassword.java
masterSDO.add(masterKey); // In PwDatabase.java
entrySDO.add(password); // In PwEntryV3.java
entrySDO.add(pass); // In EntryEditActivity.java
entrySDO.add(conf); // In EntryEditActivity.java
```

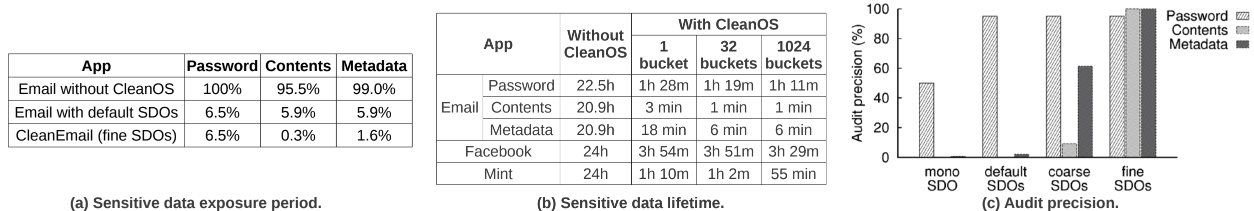
6.2 Add-on Cloud Services

CleanOS evicts sensitive data to the cloud to prevent unmediated accesses by device thieves. However, by itself, CleanOS cannot guarantee data security. For example, a thief could interact with the apps in an unlocked device or force all SDOs to decrypt. Therefore, CleanOS provides device-side mechanisms necessary for clouds to build clean-semantic security add-ons, such as assured remote wipeout or data exposure auditing. Such services already exist today (e.g., Apple's iCloud and Gmail's two-step verification), but we maintain that their semantics are unclear given the state of today's devices. We next describe an add-on service we trivially built on CleanOS.

Prototype Auditing Service. Inspired by Keypad [15], we implemented an auditing service on CleanOS. Its goal is to provide users with audit logs of what was on the device at the time of theft and what has been accessed since. The auditing service integrates with the CleanOS service and both are hosted on App Engine. When a device registers an SDO or requests a decryption key, the cloud logs that operation with the app name, SDO, and current time. In this way, the user can learn from the audit log exactly what data was leaked. For instance, Figure 5 shows a sample audit log that contains entries for SDO registration and key fetching. Were these operations to occur after the device was stolen, the user will know that the email password and KeePass entry may have been leaked.

Crucial to any auditing system is precision. In the audit log, data in different buckets of the same SDO are indistinguishable. Thus, accessing the data in one bucket may cause false alarms for evicted buckets of the same SDO. Using a finer SDO granularity helps reduce false positives. We evaluate audit precision in §7.1.

Further Examples. A cloud could build many other useful services on CleanOS. For example, the cloud could: allow its mobile users to revoke data access from their missing devices, disable access to sensitive data while the phone is outside the corporate network, and perform theft detection based on access patterns. A variety of entities would find such services useful to host. For example, a company might integrate with CleanOS on the



App	Password	Contents	Metadata
Email without CleanOS	100%	95.5%	99.0%
Email with default SDOs	6.5%	5.9%	5.9%
CleanEmail (fine SDOs)	6.5%	0.3%	1.6%

(a) Sensitive data exposure period.

App		Without CleanOS	With CleanOS		
			1 bucket	32 buckets	1024 buckets
Email	Password	22.5h	1h 28m	1h 19m	1h 11m
	Contents	20.9h	3 min	1 min	1 min
	Metadata	20.9h	18 min	6 min	6 min
Facebook		24h	3h 54m	3h 51m	3h 29m
Mint		24h	1h 10m	1h 2m	55 min

(b) Sensitive data lifetime.

(c) Audit precision.

Figure 6: **Data Exposure.** (a) Fraction of time in which sensitive data was exposed. (b) Maximum sensitive data retention period. (c) Average probability over time that tainted data was actually exposed, given that the audit log shows its SDO as exposed.

device for all corporate apps (e.g., corporate email, customer database), to access its auditing, revocation, and geography-constrained services. Similarly, Gmail could integrate with CleanOS to prevent email exposure after authentication-token revocation.

7 Evaluation

We next quantify CleanOS' security, performance, and energy characteristics. Our goal is to show that CleanOS significantly reduces sensitive data exposure while providing reasonable performance and energy consumption, even over cellular networks. We conducted all experiments on rooted Samsung Nexus S phones running CleanOS on Android 2.3.4 and TaintDroid 2.3.

7.1 Data Exposure Evaluation

To evaluate the data exposure benefits of CleanOS, we pose three questions: How much does eviction limit exposure of sensitive data? How much do default SDO heuristics limit exposure? How effective is the auditing service? To answer these questions, we recorded a 24-hour trace of one of the authors' phone running CleanOS as it was used to interact with regular apps, including Email, Facebook, and Mint. For Email, we experimented with both the unmodified app and our modified version of it, which we call CleanEmail (see §6.1). The Email app was configured with the author's personal account, which receives about ten new mails daily, and with the default 15-minute refresh period. Facebook and Mint had widgets enabled, which made them continuous services.

Sensitive Data Exposure Period. We measured the exposure period for three types of tainted data (password, content, and metadata) in the Email app. Figure 6(a) shows the fraction of time that each type of tainted data was exposed in RAM. Without CleanOS, the password was maintained in RAM all the time, and the content and metadata were exposed over 95% of the time. CleanOS reduced password exposure to 6.5%. For email content, the unmodified Email app with default SDOs reduced exposure time from 95.5% to 5.9%, and modifying the app to support fine-grained SDOs further reduced it to 0.3%. Similar observations held for metadata. To be clear, these results depend on workloads. From another, much more intensive email workload – that registered for many mailing lists and Twitter feeds – we obtained a result of 7.3% and 12.7% for content and metadata, respectively. Overall, results demonstrate a significant reduction in exposure times for tainted data. Moreover, they show that our default heuristics protect sensitive data reasonably well.

Sensitive Data Lifetime. As SDO lifetime is critical to system security, we must also examine the maximum period that a tainted object could be retained in RAM. Figure 6(b) shows the retention time for the longest-lived tainted object in three applications, where we break down email into three types. Without CleanOS, all observed applications retained certain tainted objects for more than 20 hours. With CleanOS, the maximum SDO lifetime was dramatically reduced. For instance, the Email app kept some metadata objects for as long as 20.9 hours, which CleanOS reduced to only 6 minutes when using 1024 buckets. For Facebook and Mint, the impact of bucketing on sensitive data lifetime was more limited because these apps tend to use most objects in an SDO at the same time. Overall, these results indicated that the mobile device was significantly cleaner with CleanOS.

Audit Precision. We next evaluated the effectiveness of the auditing service we built on CleanOS (see §6.2). We compared audit precision across four levels of SDO granularity in Email: (1) mono-SDO, where we marked data as only "sensitive" or "non-sensitive," (2) default SDOs, where we used default heuristics, (3) coarse SDOs, where the application defined one content SDO and one metadata SDO for all emails, and (4) fine SDOs, where each email had its own content and metadata SDOs. We define *audit precision* as the average probability over time that the tainted data is actually exposed on the device, given that the audit log shows its SDO has not been evicted.

Figure 6(c) shows audit precision for the Email app's password, content, and metadata. Password auditing was 50.0% precise with mono-SDO but increased to 95.1% with default SDOs. The content and metadata, however, had poor precision (<3%) without application support: CleanOS could not differentiate data coming from the Internet and hence added every incoming object to the *SSL* SDO. With coarse, application-specific SDOs, audit precision for email content and metadata was 9.1% and 61.3%, respectively. When fine application-specific SDOs were available, audit precision reached 100%. Thus, our default SDOs were effective in auditing password exposure, but application adaptation was needed to provide precise auditing for other types of sensitive data.

	Android 2.3.4	TaintDroid 2.3	CleanOS			
			not evicted	evicted, cached	evicted, Wi-Fi	evicted, 3G
Untainted Primitive	0.00021	0.00022	0.00026	-	-	-
Tainted Primitive	-	0.00023	0.00056	1.24	22.844	336.07
Untainted Array	0.00027	0.00029	0.00035	-	-	-
Tainted Array (S)	-	0.00030	0.00075	1.4	21.652	308.71
Tainted Array (M)	-	0.00030	0.00075	1.331	21.702	316.79
Tainted Array (L)	-	0.00030	0.00075	2.355	22.365	317.97

Figure 7: **Micro-operation Performance (milliseconds).** CleanOS Java object field access times compared with Android, TaintDroid. Times for non-sensitive and sensitive fields for various eviction states. Averages over 1,000 accesses.

7.2 Performance Evaluation

We next evaluate the performance impact of CleanOS under different workloads and networking conditions. Here, we aim to: (1) quantify raw performance overheads, (2) demonstrate that CleanOS is practical over Wi-Fi for popular apps, and (3) show how our optimizations make CleanOS practical even over slow, cellular networks. In our experience, obtaining reliable and repeatable results from the cellular network is tremendously difficult; hence, our results used emulated Wi-Fi and 3G networks with RTTs configured at 20ms and 300ms, respectively. Because our transmission units were tiny (keys were 16-byte long), we did not enforce bandwidth restrictions.

Micro-operation Performance Overheads. To evaluate raw performance overheads, we measured Java object field-access times for Android, TaintDroid, and CleanOS. Figure 7 compares them for four field types: primitives (`int`), small arrays (16 bytes), medium arrays (4KB), and large arrays (16KB). For CleanOS, we show access times both for non-sensitive fields (the vast majority) and sensitive fields under various eviction states. CleanOS' access overhead for non-sensitive fields was small compared with TaintDroid (16%), which itself was close to raw Android (6% overhead for TaintDroid). The overhead for sensitive field access increased to 141% over TaintDroid: CleanOS performed last-time-of-use bookkeeping *on every Dalvik field access instruction* (e.g., `OP_AGET`, `OP_IGET`) that involved a tainted field. Further, when evicted, CleanOS access overhead spiked dramatically, especially when the evicted field's key was not cached on the device but was fetched over Wi-Fi or 3G. Moreover, unlike in Android and TaintDroid, access times for evicted arrays in CleanOS depended on the array's size because decryption times increase with data size. For example, the "evicted, cached" column shows that decrypting a tainted array grew by 68% when the array's size increased from 16B to 16KB. Fortunately, in practice, sensitive fields are extremely rare compared with non-sensitive fields. For example, our email trace showed an average of 102,907 fields at any time, of which merely 1,889 were tainted (or 1.83%). Hence, CleanOS should acceptably affect real app performance, as shown next.

Application Performance. Figure 8(a) shows the time to launch several popular apps (i.e., bring them into the foreground) and perform typical actions, such as opening an email, viewing a KeePass entry, or loading a Web page. We chose three Web pages: a simple one (`https://iana.org/domains/example`) and two popular and more complex ones (`https://news.google.com` and `https://cnn.com`). For CleanOS, results labeled "not evicted" correspond to cases where all accessed objects were decrypted, while results labeled "evicted" correspond to cases where objects were all evicted.

In the "not evicted" case, interaction with the apps incurred a limited performance penalty compared with both TaintDroid and Android. For example, 8/13 operations incurred less than 100ms penalties over TaintDroid, and 7/13 did so over Android. Such penalties will likely go unnoticed by users, who are known to perceive delays coarsely [29]. Hence, when users interact with a recently used app, they should not feel CleanOS' presence.

When users interact with a cold app ("evicted" columns for unoptimized CleanOS), however, performance degraded but remained usable for Wi-Fi networks. Our cheapest app is the browser, for which CleanOS incurred 8-23% overheads over Android for all operations. The reason is two-fold: (1) the browser deals with little sensitive data, and (2) during page loads, the browser fetches large amounts of data over Wi-Fi, which dwarf CleanOS' key traffic delays. The most expensive app for CleanOS is CleanEmail, which incurred a larger penalty than Email for "evicted" launches due to more granular tainting. For example, while Email needed to fetch 2 keys to load an email, CleanEmail needed to fetch 3 keys.

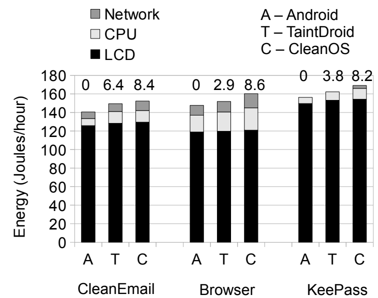
Over 3G, CleanOS penalties after eviction became significant. While some operations remained within reasonable bounds (e.g., launching the browser and loading `iana.org` or `cnn.com`), many operations incurred overheads in excess of 100%. For example, loading an email onto the screen jumped from 197ms to 1.1s for Email and 1.4s for CleanEmail. Such delays likely affect usability.

Effect of Optimizations on Application Performance. Column "Optimized CleanOS" in Figure 8(a) shows the elapsed time of repeat operations under our bulk prefetching optimization (see §5). All timed operations were invoked in the previous application session; therefore, all of their relevant keys were prefetched together as part of one bulk request during the timed session. The results show dramatic improvements in performance for both launching and interacting with the apps. For example, CleanEmail – our most expensive application – launched in 589ms over 3G compared with 919ms on unoptimized CleanOS (35.9% improvement) and loaded a previously read email in 420ms compared with 1.4s (71% improvement). In general, this optimization lets an app re-launch incur little more than one RTT over non-evicted CleanOS, while subsequent repeat operations incur no RTT. Natu-

Application	Action		Android 2.3.4	TaintDroid 2.3	CleanOS			Optimized CleanOS
					not evicted	evicted, Wi-Fi	evicted, 3G	evicted, 3G*
Email	Launch		197	202	241	312	919	589
	Read Message		212	254	387	501	1165	379
CleanEmail	Launch		-	-	291	315	902	598
	Read Message		-	-	452	526	1472	421
KeePass	Launch		173	192	217	221	527	672
	Read Entry		125	150	146	155	479	135
Browser	Launch		130	151	160	144	222	138
	Load Page (iana)	Wi-Fi	488	483	658	605	-	-
		3G	2067	2114	2125	-	2136	2031
	Load Page (GNews)	Wi-Fi	1072	1043	1270	1160	-	-
		3G	1717	2475	2475	-	3536	2942
	Load Page (CNN)	Wi-Fi	1065	1136	1394	1446	-	-
		3G	4570	4709	4325	-	4619	4538

* Actions were performed before.

(a) App Performance (milliseconds).



(b) Energy over Wi-Fi.

Figure 8: **Application Performance and Energy Consumption.** (a) Performance of various popular app activities under Android, TaintDroid, and CleanOS for various eviction states and configurations. Results are averages over 40 runs. (b) Hourly energy consumption attributed by PowerTutor to the three apps when running a long-term synthetic workload for at least 3 hours. Numbers on top of each bar show energy overhead over default Android in percent.

rally, our optimization will not benefit non-repeat operations, such as loading a brand new or long-unread email. However, one type of operation that will always benefit is app launch, a latency-sensitive operation on mobiles.

Despite their performance benefits, our optimizations may increase data exposure. When applying these optimizations to the workloads in §7.1, we obtained limited, but non-trivial, exposure impact. The period for each type of tainted data increased by up to 0.9 percentage points for the workload in Figure 6(a), and by up to 23.2 percentage points for our intensive Email workload. Prefetching keys from multiple sessions would cause further exposure. Hence, CleanOS should best apply this optimization only in specific cases (e.g., over 3G).

Overhead Estimation for SDO Stable Storage Extension. Thus far, our results show CleanOS' overheads for eviction of *in-RAM SDOs*. While we have not fully implemented the SDO extension to stable storage, we now offer rough estimates for the extra overheads to expect from such an extension. We expect the major sources of overhead to be: (1) the key fetches required to access encrypted database items, and (2) the extra encryption/decryption that occurs when accessing these items. To account for (1), we ran experiments with our test applications that instruct CleanOS to fetch the appropriate decryption keys for any tainted database items being accessed. To account for (2), we added an extra 20% overhead per query, a number reported by CryptDB [33], which also does per-item encryption.

With this methodology, we estimate that extending SDOs to SQLite would result in additional overheads ranging between 0-65% on 3G over CleanOS with in-RAM SDOs. We predict that these operations will suffer the most: KeePass Launch (869ms, or 64.9% additional overhead), CleanEmail Read (1887ms, or 28.2% additional overhead), and Browser Load (2542ms, 4086ms, and 4573ms for `iana.org`, `news.google.com`, and `cnn.com`, respectively, or 15-19% additional overhead). Most of these overheads (82-99% across all apps) are due to extra RTTs incurred by necessary key fetches, which are optimizable via batch prefetching. Thus, overall, we believe that our system will be practical from a performance perspective even when implemented in full.

7.3 Energy and Network Evaluation

CleanOS' encryption, network traffic, and extra GCs raise concerns about its impact on energy consumption. To evaluate this impact, we ran coarse-grained experiments that drove a simple, long-term workload against each app (CleanEmail, Browser, and KeePass) using MonkeyRunner [17] and measured consumption using the PowerTutor online power monitor [42]. The workload repeatedly launched an app, performed a set of typical tasks (such as reading emails, accessing entries in KeePass, and visiting Web pages in the browser), sent the app into the background, and then slept for 15 minutes. Each app interaction lasted for 36-46s, after which we promptly turned off the LCD. We ran the workload continuously for at least 3 hours and plotted per-app power consumption as reported by PowerTutor.

Figure 8(b) shows energy consumption for Android, TaintDroid, and CleanOS over a *real* home Wi-Fi network. For each app, we show the energy consumed by the LCD, CPU, and Wi-Fi. Results show that CleanOS' total energy overheads over Wi-Fi were small compared with both Android and TaintDroid: 8.2-8.4% over Android (see labels above bars) and 1.9-5.5% over TaintDroid. Drilling down on resource overheads, we observe that CleanOS increased energy consumption of both the network (44-45%) and the CPU (32-74%), but those overheads were dwarfed by the LCD energy draw. In general, our overheads were smallest for the browser, which itself consumed relatively more CPU and network energy, and largest for KeePass, a lightweight application that performed little computation and had no network traffic.

Over 3G, energy overheads due to network traffic will likely increase. Our experience shows that experiment-

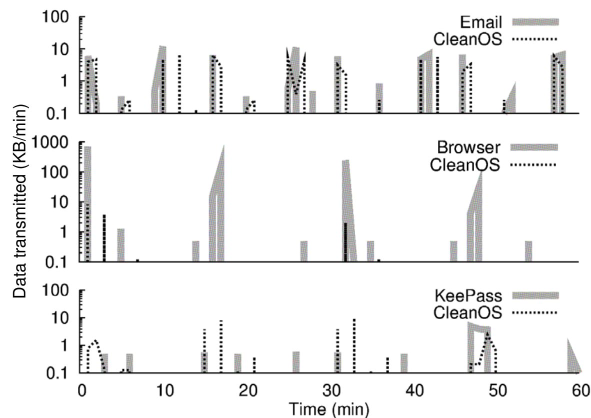


Figure 9: **Network Traffic Patterns of Apps vs. CleanOS.** CleanOS traffic vs. app traffic for a one-hour trace. The Y axis is in log scale. In our cases, the phone has background traffic, which is included in both app and CleanOS lines.

ing with 3G networks leads to very unstable and unrepeatable results; hence, for these networks, we rely on an analytic evaluation grounded in a study of CleanOS' network traffic. Figure 9 compares CleanOS' traffic patterns to those of the three apps using one-hour traces from our energy experiments. It shows that CleanOS' network consumption depends on the application's own network profile. For networked apps, such as email and browser, CleanOS' traffic closely follows the app's own traffic distribution over time. For example, for email, of the 24 minutes during which CleanOS issued some traffic, only 9 of those had no accompanying app traffic; for the browser, only 1 out of 5 one-minute periods did so. From an energy perspective, this means that CleanOS usually piggybacks on the app's own use of the network and only rarely needs to hold the interface up for its own purposes. On the other hand, for local-only apps, such as KeePass, CleanOS uses the network mostly for its own purpose; but even in such cases, however, its traffic will be rare, brief, and small ($\leq 10KB/min$). Thus, we expect CleanOS to be practical from an energy perspective.

8 Security Discussion and Limitations

We now discuss CleanOS' security implications and limitations. There are two types of data that an attacker might seek: unevicted data and evicted data. CleanOS does not protect unevicted data on a stolen device; instead, it seeks to minimize the amount of such data. An audit-enabled cloud service can provide users with a robust audit trail of data exposed at the time of loss and data retrieved since. For evicted data, clouds can do much more. For example, after theft has been detected, they can revoke the device's access to still evicted data. They can also monitor accesses to keys to detect anomalous behavior.

A thief might also try to retrieve keys for all evicted SDOs before the cloud disables them. Such aggressive attackers could be identified via anomalous access-pattern detection. To evade detection, the attacker could retrieve SDO keys only for objects of interest, such as emails with tempting subjects. While some attackers may be unwilling to do so for fear of revealing their identities, the cloud can provide an audit log of such accesses.

Attackers might also attempt to break the disconnection password to hoard keys for apps of interest without raising suspicion. CleanOS could enforce sufficient entropy to make the disconnection password, which is extremely rarely used, much stronger than a regular password (which a user must type every time he unlocks his device). However, even if the password were broken, the cloud could provide evidence of the attacker's behavior.

Adversaries may perform network attacks to sniff or disrupt CleanOS device-cloud traffic. To prevent sniffing of keys from network traffic, we encrypt connections and authenticate the device to the cloud using a pre-established secret key (akin to the device token in Gmail's two-factor verification) and the cloud using public key cryptography. An attacker could also disrupt CleanOS device-cloud communication to induce CleanOS into an accumulation mode, where it defers eviction until cloud connectivity returns. To defend, CleanOS bounds its eviction delay for temporary disconnections. Moreover, a thief could prevent eviction messages from arriving at the cloud. However, dropping those messages will not affect confidentiality since data eviction will complete as planned, but it might raise auditing false positives.

One CleanOS limitation is its limited coverage outside the Java realm. To be clear, expunging sensitive data from Java is an important contribution: 9/14 apps in Figure 2(b) would expose some sensitive data permanently in RAM if we did not do so. Moreover, we have incorporated some basic multi-level secure deallocation techniques and have modified two popular native libraries to limit exposure (SQLite and WebKit). However, any data retained in other buffers or caches in the OS or native libraries remains exposed. To limit this exposure, we recommend: (1) incorporating additional OS data scrubbing mechanisms [10], (2) inspecting all remaining system libraries for caches as we do for SQLite and WebKit, and (3) either disabling all third-party libraries (an approach similar to TaintDroid's [12]) or informing the cloud about any data leakages to uninspected third-party libraries.

9 Related Work

CleanOS builds upon prior work that we now describe.

Encrypted File Systems. Encrypted file systems [11] and full-disk encryption [26, 38] are designed to protect data stored on a vulnerable device, but they do not protect data in RAM. Moreover, as discussed in §3 (Threat Model) and in prior work [15, 41], these systems can fail in the real world due to human factors (e.g., non-existent or poor passwords) and physical attacks (e.g., key re-

trieval from RAM via cold-boot attacks [19]). CleanOS recognizes these limitations and promptly removes unused data from the vulnerable device.

Encrypted RAM Systems. Encrypted RAM systems – such as XOM [23], CryptKeeper [31], and encrypted swap [34] – encrypt data while it sits in RAM. CryptKeeper resembles the CleanOS model by encrypting all memory pages except for a small working set, thereby achieving a similar encrypted-unless-in-use effect as CleanOS. However, while the data is encrypted in these systems, the decryption keys themselves are still available in RAM and potentially accessible to memory-harvesting unless extra hardware is deployed. Moreover, if the device were unlocked or the thief found the user's password, encrypted RAM would have no effect.

ZIA [7, 8] encrypts mobile data in RAM and on disk whenever a device is not near its owner. The user wears a beaconing token at all times, whose presence is detected by the mobile. Like ZIA, CleanOS encrypts data after a period of non-use, but the granularities, method, and usage model are different. For example, we disable unused data at the Java object level as opposed to the device level, evict data to clouds for increased post-theft control, and do not require users to carry (and secure!) tokens.

Mobile Wipe-Out Systems. Varied commercial wipe-out systems exist and help increase users' post-theft data control. For example, remote wipe-out systems, such as iCloud [3], let the users send "kill" messages to lost devices. Unfortunately, these systems require network connectivity to function correctly. If the thief prevents device connectivity (e.g., by wrapping it into a Faraday cage), the device will not receive the message and therefore not complete its wipeout. Moreover, configuring the device to self-destruct after a number of failed authentication attempts helps prevent access to file system data, but it does not preclude memory harvesting attacks, such as coldboot imaging [19]. Such attacks are particularly problematic on mobile devices, which hardly ever power off.

Cloud-based Mobile Security Services. The value of the cloud for increased data control is being increasingly recognized. Examples of cloud-based security services include: online data access revocation with two-step verification [18], location-based access control with location-aware encryption [37], and cloud-based authentication with capture-resilient cryptography [25]. Generally, these systems prevent the compromise of data not already exposed on the device, but they do not guarantee security for mobile-resident data. For example, none of these systems takes RAM-resident data into account, and the Google two-step verification does not even consider storage. CleanOS cleanses device RAM and storage in support of such security services.

Keypad. Particularly relevant is Keypad [15], an auditing file system for old-generation mobile devices, such as laptops and USB sticks, that achieves file-level, strong-semantic auditing. CleanOS shares Keypad's threat model, and our auditing service was inspired by it. However, in addition to its support for in-RAM data auditing, CleanOS also differs from Keypad in its focus on new-generation mobile technologies, such as Android, which have distinct auditing granularity requirements. For example, file-level auditing in Keypad would be ineffective for apps using the SQLite database since they all would be stored within one single file. Instead, CleanOS defines SDOs, an abstraction that encompasses fine-grained objects, database items, and sdcard files.

Secure-Deletion Systems. Secure deletion has been recognized as a key OS primitive. It erases data in memory [6], OS buffers [10], and stable storage [30, 39, 4] once the data is not needed by the application. CleanOS explicitly assumes the existence and robustness of such systems, but addresses a distinct, important part of the sensitive data exposure problem for the first time: securing data explictly hoarded by applications for performance or convenience. CleanOS SDOs resemble the self-destructing data abstraction in Vanish [16] in that they "disappear" over time, but the setting is different: Vanish makes Web data disappear after a specified time post-creation, whereas SDOs make mobile data disappear if they are unused for a specified time.

10 Conclusions

This paper described CleanOS, a new design for the Android OS that manages sensitive data rigorously and keeps mobile devices clean at any point in time. Unlike Android, which lets sensitive data accumulate in cleartext RAM and on disk, CleanOS eliminates it from the vulnerable device by evicting it to the cloud whenever it is not needed on the device. It provides a clean-semantic foundation for clouds to build add-on services, such as data access revocation after a device has been lost or post-theft data exposure auditing. We implemented CleanOS by instrumenting Android's Java virtual machine to securely evict sensitive data objects after a specified period of non-use. On top of CleanOS, we built a sample auditing cloud service. Our experiments demonstrate that CleanOS limits data exposure significantly while imposing acceptable performance overheads and offering sound semantics for cloud-based applications.

11 Acknowledgements

We thank our shepherd, Petros Maniatis, and anonymous reviewers for their valuable comments. We also thank Steve Gribble, Angelos Keromytis, Hank Levy, Simha Sethumadhavan, Salvatore Stolfo, and Junfeng Yang for their feedback. This work was supported by DARPA through FA8650-11-C-7190 and FA8750-10-2-0253 and NSF through CNS-0905246.

References

[1] R. Anderson and M. Kuhn. Tamper resistance: A cautionary note. In *Proc. of the USENIX Workshop on Electronics Commerce*, 1996.

[2] Android Developers Blog. Avoiding memory leaks. android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html, 2009.

[3] Apple iCloud. Find my iPhone, iPad, and Mac. www.apple.com/icloud/features/find-my-iphone.html, 2012.

[4] D. Boneh and R. Lipton. A revocable backup system. In *Proc. of USENIX Security*, 1996.

[5] S. Chen, M. Kozuch, T. Strigkos, and et.al. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[6] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of USENIX Security*, 2005.

[7] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proc. of the ACM Annual International Conference on Mobile Computing and Networking*, 2002.

[8] M. D. Corner and B. D. Noble. Protecting applications with transient authentication. In *Proc. of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[9] L. P. Cox and P. Gilbert. Redflag: Reducing inadvertent leaks by personal machines. Technical Report TR-2009-02, Duke University, 2009.

[10] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] EncFS. www.arg0.net/encfs, 2010.

[12] W. Enck, P. Gilbert, B. gon 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 *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[13] Federal Communications Commission. Announcement of new initiatives to combat smartphone and data theft. www.fcc.gov/document/announcement-new-initiatives-combat-smartphone-and-data-theft, 2012.

[14] Future of Privacy Forum, Center for Democracy & Technology. Best practices for mobile applications developers. www.futureofprivacy.org/wp-content/uploads/Apps-Best-Practices-v-beta.pdf, 2011.

[15] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[16] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security*, 2009.

[17] Google Inc. MonkeyRunner. developer.android.com/tools/help/monkeyrunner_concepts.html, 2012.

[18] Google Inc. Two-step verification. support.google.com/accounts/bin/topic.py?hl=en&topic=28786, 2012.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of USENIX Security*, 2008.

[20] Imperva. Consumer password practices. www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf, 2010.

[21] Intel Corporation. Laptop security with Intel Anti-Theft technology. www.intel.com/content/www/us/en/architecture-and-technology/anti-theft/anti-theft-general-technology.html, 2012.

[22] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). tools.ietf.org/html/rfc5869, 2010.

[23] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[24] Lookout Mobile Security. Lost and found: The challenges of finding your lost or stolen phone. blog.mylookout.com/blog/2011/07/12/lost-and-found-the-challenges-of-finding-your-lost-or-stolen-phone, 2011.

[25] P. MacKenzie and M. Reiter. Networked cryptographic devices resilient to capture. In *Proc. of USENIX Security*, 2001.

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

[27] Microsoft Corporation. Create strong passwords. www.microsoft.com/security/online-privacy/passwords-create.aspx, 2012.

[28] M. Milian. U.S. government, military to get secure Android phones. www.cnn.com/2012/02/03/tech/mobile/government-android-phones/index.html, 2012.

[29] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[30] R. Perlman. File system design with assured delete. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[31] P. A. H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Proc. of the IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[32] Ponemon Institute. The lost smartphone problem. www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf, 2011.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[34] N. Provos. Encrypting virtual memory. In *Proc. of USENIX Security*, 2000.

[35] J. Robertson. Security chip that does encryption in PCs hacked. www.usatoday.com/tech/news/computersecurity/2010-02-08-security-chip-pc-hacked_N.htm, 2010.

[36] M. Savage. NHS 'loses' thousands of medical records. www.independent.co.uk/news/uk/politics/nhs-loses-thousands-of-medical-records-1690398.html, 2009.

[37] A. Studer and A. Perrig. Mobile user location-specific encryption (MULE): Using your office as your password. In *Proc. of the ACM Conference on Wireless Network Security (WiSec)*, 2010.

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

[39] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. FADE: Secure overlay cloud storage for file assured deletion. In *Proc. of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[40] W3C. Mobile app best practices. www.w3.org/TR/mwabp, 2010.

[41] A. Whitten and J. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proc. of USENIX Security*, 1999.

[42] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2000.