Deadbolt: Locking Down Android Disk Encryption*

Adam Skillen
Carleton University
Ottawa, Canada
askillen@ccsl.carleton.ca

David Barrera
Carleton University
Ottawa, Canada
dbarrera@ccsl.carleton.ca

Paul C. van Oorschot
Carleton University
Ottawa, Canada
paulv@scs.carleton.ca

ABSTRACT

Android devices use volume encryption to protect private data storage. While this paradigm has been widely adopted for safeguarding PC storage, the always-on mobile usage model makes volume encryption a weaker proposition for data confidentiality on mobile devices. PCs are routinely shut down which effectively secures private data and encryption keys. Mobile devices, on the other hand, typically remain powered-on for long periods and rely on a lock-screen for protection. This leaves lock-screen protection, something routinely bypassed, as the only barrier securing private data and encryption keys. Users are unlikely to embrace a practice of shutting down their mobile phones, as it impairs their communication and computing abilities. We propose Deadbolt: a method for maintaining most mobile computing functionality, while offering the security benefits of a powered off device with respect to storage encryption. Deadbolt prevents access to internal storage even if the adversary can exploit a lock screen bypass vulnerability or perform a cold boot attack. Users can gracefully switch between the Deadbolt and unlocked modes in less time than a system reboot. Deadbolt offers the additional benefit of an *incognito* environment in which logs and actions will not be recorded.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Privacy Protection; E.3 [**Data**]: Data Encryption

Keywords

Disk Encryption; Lock-screen; Cold boot attack

1. INTRODUCTION

Due to the sensitive nature of data stored on mobile devices, all major mobile OSes now offer some form of storage encryption. Google introduced full disk encryption (FDE) in Android 3.0 to encrypt the entire `userdata` partition. Some OSes (*e.g.,* BlackBerry) employ per-file encryption, to protect only privacy sensitive data. Other vendors (*e.g.,* Apple, Microsoft) leverage the use of on-device specialized hardware to support encryption.

FDE traditionally operates below the filesystem to provide on-the-fly decryption and encryption for every read/write operation on the block device. For this reason, the encryption key must always be available while the filesystem is mounted. When the device is in this *unlocked* state, the key and data are both susceptible to disclosure (*e.g.,* resident malware or the cold boot attack [12]). FDE is used extensively in the PC realm (*e.g.,* Microsoft BitLocker, Mac FileVault, and LUKS for Linux), however the mobile landscape presents a new set of challenges. To protect the key and encrypted data, a PC can be shut down or put into hibernation mode (assuming swap space or hibernation files are also encrypted). In both cases, the key is removed from memory and data is secure.

Mobile devices, on the other hand, are rarely shut down, and are instead *screen-locked*[1] or *PIN-locked* to protect user data while still providing basic communication functionality (*e.g.,* telephony, SMS, network connectivity). Indeed, the always-on mobile usage model makes FDE a weaker proposition for data confidentiality on mobile devices; the desired security benefits of FDE are not truly achieved while the device is operational, and shutting off the device eliminates the communication utility afforded by the device.

Solutions have been proposed to increase the security of software-only FDE by avoiding the use of RAM to store keys (see Section 6). These proposals, while generally effective against cold boot attacks, must still keep encrypted data unlocked while the device is powered on. If the device is obtained in this state by an adversary, data is still vulnerable to disclosure in the event of the lock-screen being bypassed, or direct hardware manipulation.

In this paper, we introduce a method for providing enhanced data protection beyond that which is currently realized by the combination of Android FDE with a lock-screen. Android users are currently forced to decide between functionality and security. If the device remains powered on, the user can receive calls and notifications but the FDE key is in memory. If the device is powered off, users get the full protection of encryption but both lose the functionality and incur the latency of booting up every time they want to use it. Our proposal, Deadbolt, protects the FDE key and encrypted data without losing communication functionality or the need to incorporate additional hardware. Deadbolt gives the protection of a device that is powered off, and the functionality of a device that is powered on, as well as a diminished delay in returning the device to a fully functional state.

Deadbolt is meant to complement the Android lock-screen in situations when additional protection is necessary to contend with a skilled and motivated adversary. Under normal

*Author Copy August 22, 2013

[1]We use the term lock-screen or screen-locked throughout the paper to refer to keeping the device powered-on, but requiring a PIN, password or other user secret to resume interaction with the device.

circumstances, the lock-screen and FDE may be enough to discourage a casual antagonist from further attempting to access the user's data (*e.g.,* at their place of employment). In higher-threat situations (*e.g.,* traveling home from work), the device could be captured and subjected to more rigorous attacks. In this situation, users should invoke Deadbolt (either manually or automatically) to protect their FDE key and encrypted data even in the event the lock-screen is bypassed. While in the Deadbolt environment, the user can continue to make phone calls, send text messages, and browse the web via a mobile data network.

Contributions. We design, implement and evaluate a software-only method for protecting the FDE key and encrypted user data, while still providing basic mobile smartdevice (*e.g.,* smartphone, tablet) functionality. Our key insight is that all user data need not be unlocked at all times, especially when the device is not being actively used (*e.g.,* in a bag or pocket). When the device is in Deadbolt mode, it is resilient to cold boot attacks and lock-screen bypass vulnerabilities. Additionally, because system partitions are mounted and unmounted on the fly, resuming FDE mode after leaving Deadbolt mode is faster than a full restart of the device.

The remainder of the paper is organized as follows. Section 2 covers background on mobile device encryption approaches including Android-specific details. Section 3 describes the design and implementation of Deadbolt. Section 4 provides a performance evaluation and security analysis. In Section 5 we discuss enhancements and limitations of our proposal. Section 6 explores related work. We conclude in Section 7.

2. BACKGROUND

This section provides background on mobile device storage encryption techniques. We also provide an overview of the Android system and FDE implementation specifics.

2.1 Encrypted Storage on Mobile Platforms

For encryption, most manufacturers use either file-based (*e.g.,* iOS, BlackBerry) or full-disk encryption (*e.g.,* Android, Windows Phone). The advantage of file-based encryption over FDE is that the storage encryption keys can be wiped from RAM when the screen is locked without losing functionality. However, FDE provides transparent protection for all data stored on the device (*i.e.,* no special actions or routines are required by the user or app developer to encrypt stored data).

Apple. Apple iOS devices use a hardware crypto co-processor physically between the storage and RAM [2]. The processor contains a burned-in device unique ID (UID) encryption key. All data stored on the device is, at minimum, encrypted with the UID derived key. The dedicated crypto engine effectively keeps all key material out of RAM and ties the encrypted storage to a particular device (*i.e.,* chip-off attacks would amount to brute-forcing the AES key instead of the password [2]). Per-file keys are generated for each file stored on the device. The file keys are stored in encrypted meta-data and used to encrypt file contents. File keys are wrapped using the AES-key wrap algorithm [27] with either a UID derived key, or a UID and password derived key. Whether the password is used depends on the situation – *e.g.,* files that may only be accessible after user authentication require a UID and password derived key. This setup allows the OS to respond to events such as phone calls and notifications without access to any password protected content. App developers must explicitly call the encryption API to protect app data with a password derived key, otherwise the data is only protected with the UID key [34].

Blackberry. BlackBerry devices can enable the *content protection* feature to encrypt both internal and removable storage [24]. Content protection does not use FDE, and instead only certain data will be encrypted (*e.g.,* emails, contacts). Content protection uses a password derived key-encryption-key (KEK) to encrypt a storage AES data-encryption-key (DEK), and a message encryption public/private key pair. These keys are encrypted with the KEK and stored in flash. When a device is screen-locked, the DEK and message private keys are wiped from RAM. Any messages received while the device is locked are encrypted with the message public key, and decrypted after unlocking the private key. Encryption and decryption are performed on-the-fly when the storage is accessed. As the keys are stored in flash, a *chip-off* technique could be used to brute-force the password and recover the encrypted data.[2] BlackBerry devices also use a memory cleaning enhancement to the Java garbage-collector, to wipe plaintext fragments from RAM after a period of inactivity [24].

Windows Phone 8. Windows Phone 8 provides storage encryption based on BitLocker for the OS and internal user data partitions [17]. BitLocker provides FDE-like filesystem encryption [9]. Device encryption is enabled by mobile device management policy (*e.g.,* from Exchange ActiveSync server) as opposed to the device's local settings. According to the Microsoft Windows Phone 8 security overview, the encryption key is protected by the TPM [17]. Use of a TPM to store the encryption key avoids the need to rely on a user secret to protect the key (*i.e.,* the user does not need to enter a PIN or password for pre-boot authentication). Despite the use of a TPM, the encryption key must remain in RAM while the screen is locked for the software implementation of BitLocker. While the key is in RAM, the encrypted data is vulnerable to cold boot and lock-screen bypass attacks (see Section 3.2).

2.2 Android System

This section details Android's FDE implementation, and associated OS components.

Android's FDE implementation. On Android, the `userdata` partition holds all user-installed apps and user-created data. The `userdata` partition is the only partition that is encrypted in Android's FDE implementation, which uses the Linux kernel device-mapper crypto target (`dm-crypt` [7]). The kernel and OS partitions are generally mounted read-only, so no private user data will be stored in these locations. Removable storage (if available) is not encrypted. Enabling FDE on Android requires that the user set a lock-screen PIN or password (*i.e.,* pattern and *face unlock* secrets may not be used).

A randomly chosen master volume key is used to decrypt and encrypt data on-the-fly for any read/write operation performed on the `userdata` storage [1]. The AES-128 cipher is used in the CBC mode with ESSIV derived IVs. The user's screen unlock password is used to derive a key-encryption-key (KEK) using 2000 iterations of the PBKDF2 [14] function.

[2] `http://www.binaryintel.com/chip-forensics-device/`

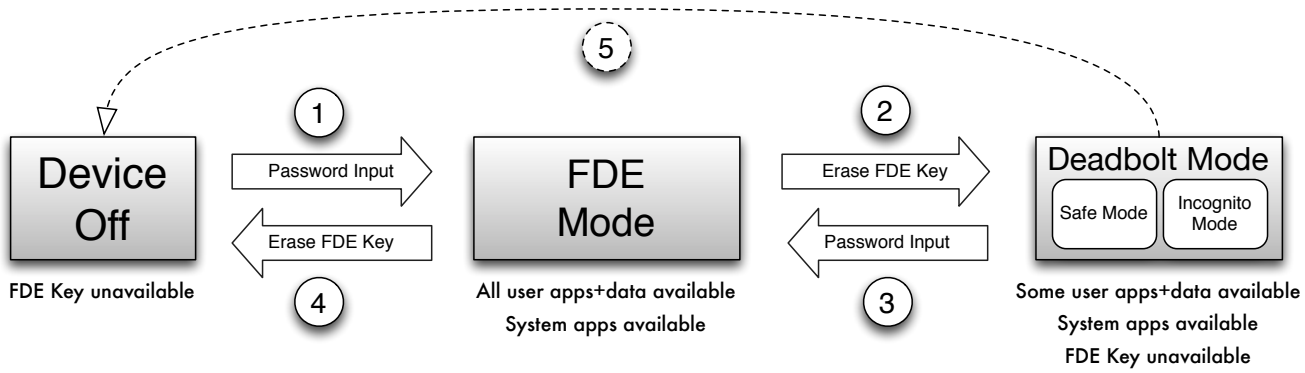


Figure 1: Overview diagram of state transitions in the Deadbolt architecture. For each state, we note advantageous security and usability properties. All transition steps are user-initiated (unless specified by policy, see Section 3.3). Steps 1 and 3 require user input to generate the password-derived FDE key, which is erased in steps 2 and 4. Step 5 is optional.

The KEK is used to encrypt the master volume key, which is then stored in a volume footer located immediately following the filesystem.

On system boot, the user is prompted for their password before the entire Android framework is initialized. Their password is used with PBKDF2 to regenerate the KEK which in turn is used to decrypt the master volume key. The `userdata` partition is mapped with the master volume key and mounted onto the filesystem at `/data`. The master volume key remains in RAM until the phone is shut off or rebooted [1].

Since the lock-screen secret is also used to protect the volume key stored in the footer, decrypting the storage is reduced to brute-forcing this secret (note that both lock-screen passwords and PINs are limited to 16 characters). Furthermore, even when FDE is enabled, bypassing the Android lock-screen provides access to unencrypted user data (such vulnerabilities are not uncommon; see Section 3.2).

Android base install. Android devices ship with a base OS image that includes a set of applications providing basic smart-device functionality. The open source release of Android (AOSP) includes applications for contacts, web browsing, calendar, image gallery, music player, and others. When the target build device is a smartphone, additional phone-specific apps are built, such as an app for sending SMSs, making phone calls, and configuring the cellular data network. Commercial Android releases include a suite of Google apps with further base functionality (*e.g.,* maps, Google Play Store, *etc.*). These apps are on a read-only partition of the device, but user data for the apps is stored on the `userdata` partition.

3. DEADBOLT

This section describes our software-only method of protecting the Android FDE key, while maintaining core smart-device functionality.

3.1 Overview

As mentioned, Deadbolt is meant to complement the default Android lock-screen mechanism. When the user is in a low risk environment, and their device will not be subjected to rigorous attack (*e.g.,* at home or work) they can use the lock-screen to protect their device. When the possibility exists that the device may be lost or confiscated and subjected to higher attack (*e.g.,* when commuting or traveling), they should enable Deadbolt to further protect their private data.

Invoking Deadbolt pauses the running Android framework, unmounts the encrypted storage, and removes the FDE key from RAM (see step 2 in Figure 1). The user is delivered into a functional Android environment in which the user's data volume is encrypted and the encryption key is discarded. Thus, no private user data is accessible. The base set of Android applications (see Section 2.2) are available for use, and an empty `userdata` volume is mounted as a RAM filesystem (`tmpfs`). This temporary instance of Android allows the user to send and receive phone calls and text messages, and use the cell phone data network. Although the base apps themselves are available from within the temporary environment, the associated user data is not (*e.g.,* the phone dialer app is available, but the user's contacts and call logs are not). In the default configuration, none of the user's after-market apps are available in the temporary environment. The Deadbolt environment can be initialized in one of two modes:

- Mobile *incognito* environment (default): all logs, files, and activities are discarded when the user exits the temporary environment, by default (*cf.* modern browser's private/incognito mode). Users may optionally import some data from the encrypted `userdata` volume (*e.g.,* Wi-Fi passwords, contacts, *etc.*; see Section 3.4) into the temporary environment, before invoking Deadbolt. Users may also choose to copy certain changes back to the encrypted volume (*e.g.,* call log, received SMS messages, *etc.*) before ending the Deadbolt session. This provides the user with the option of performing some actions deniably (*i.e.,* without maintaining any logs or records on the device; *cf.* [8]). However, not unlike a browser's private/incognito mode, the cell carrier or ISP may maintain activity records and private data could otherwise be captured off-device (see *e.g.,* [30]).

- Mobile *safe-mode* environment: Deadbolt can alternatively be invoked in a mobile safe-mode environment

in which no data may be copied to/from the encrypted storage. The user may wish to install an untrusted app (*e.g.,* from a 3rd party app market) or visit an insecure website in this mode. Similar to incognito mode, none of the user's private data is accessible in this mode. Thus, even a privileged app with access to the root file system cannot compromise the confidentiality of the user's data (as the encrypted `userdata` volume is unavailable). All of the changes made during the present Deadbolt session (including app installations) are discarded when the user resumes their encrypted environment. Note that malware which can obtain access to the read-only `/system` volume may be able to stay resident when the Deadbolt session ends (see *e.g.,* [22]). The Deadbolt safe-mode can protect against over-privileged apps and leakage of private user data.

3.2 Threat Model

The main attack that Deadbolt protects against is bypassing the lock-screen to access the internal storage. The many incarnations of this attack can be divided into:

1. Exploiting software vulnerabilities. Several vulnerabilities have been discovered which can allow an adversary to bypass the lock-screen without the user's secret (*e.g.,* a recent Skype bug[3]). There are also lock-screen bypass apps that can be pushed onto the device if the adversary has the user's Google credentials.[4] Potential bugs in network services or the Android debug bridge (`adb`) could allow the adversary to access the internal storage without the unlock secret.

2. Cold boot attacks. The underlying `dm-crypt` system employed by Android maintains keys and intermediate state in RAM, and as such is susceptible to the cold boot attack (see Section 6). Müller *et al.* recently demonstrated a successful cold boot attack against Android FDE [20].

Deadbolt protects against lock-screen bypass vulnerabilities by unmounting the encrypted `userdata` partition, and securely deallocating (*i.e.,* overwriting with zeros) the key from RAM. Additional security benefits may be achieved by attempting to scrub all sensitive plaintext fragments from RAM (see Section 4.2). Deadbolt does not protect against offline password guessing attacks on the stored volume key. Such attacks are out of our present scope.

We assume the adversary is capable of obtaining physical access to the device while it is in Deadbolt mode. As such, any attacks requiring physical access are within scope.

Intended users. Deadbolt is designed for security conscious users who have already enabled (or wish to enable) full disk encryption on their device. We also consider enterprise users who have been required to enable FDE on their devices (*e.g.,* due to corporate policy). Since these users may not always be experts, enterprise device administrators may wish to require Deadbolt-like functionality for increased data protection. For non-experts, automatically enabling Deadbolt after a period of inactivity may be preferred (see Section 3.3 under the user interface heading).

[3] `http://seclists.org/fulldisclosure/2013/Jul/6`
[4] `http://android.m.brothersoft.com/screen_lock_bypass-115258.html`

3.3 Implementation

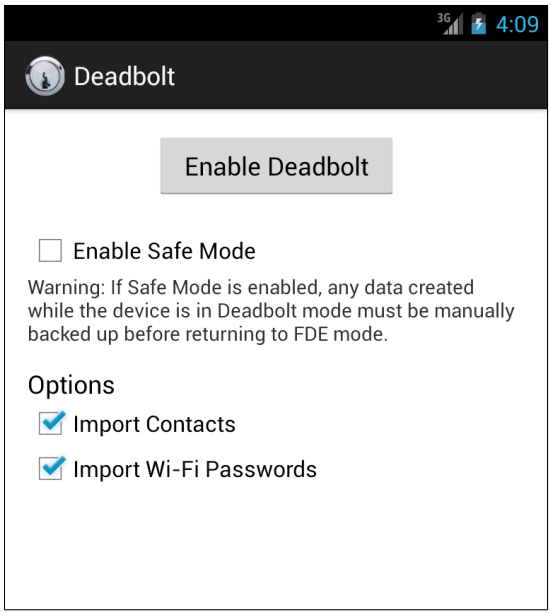
We developed and tested Deadbolt on an Asus Nexus 7 tablet using the AOSP 4.2.2 source code. We managed to incorporate our functionality into the Android volume mounting daemon (`vold`) in less than 400 additional lines of code. The volume mounting daemon was the logical component to enhance, to provide Deadbolt functionality, as it is already tasked with staging and mounting encrypted volumes on the device. Additional minor changes were made to the Android mount service to enable inter-process communication between Dalvik system apps and vold.

Changes to `vold` subsystem. The Android framework classifies services into 3 categories: core, main, and late start. When an FDE-enabled device is powered on, a minimal set of services are started to request the user's password, which is used to derive the FDE key (see Section 2.2). The encrypted volume is then mounted on `/data` and the framework is restarted with all services running. Android runs a system properties service to maintain information about the system state. The properties service stores key-value pairs in memory. The init process monitors the property `vold.decrypt` to determine which classes should be started or stopped. We leverage this behavior to suspend and restart the framework when unmounting the encrypted volume.

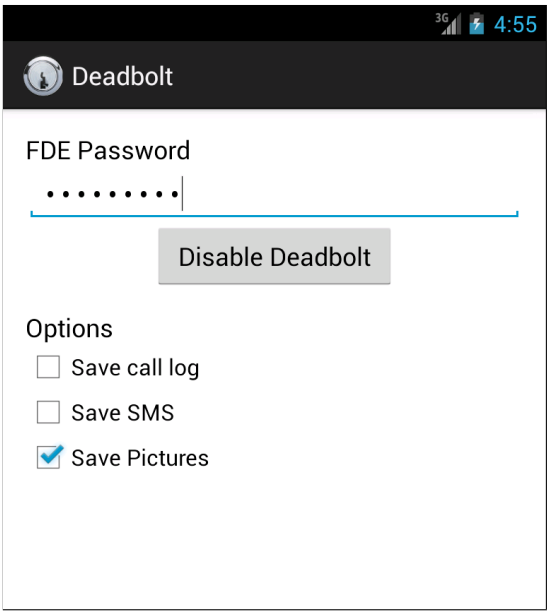
Two functions were added to the `vold` `cryptfs` subsystem: *cryptfs lock* and *cryptfs unlock*. During the operation of *cryptfs lock*, the `vold.decrypt` property is set to `trigger_shutdown_framework` which instructs init to stop all services in the main and late start classes. Some of the services in these classes (*e.g.,* Zygote – the parent Dalvik virtual machine that spawns all other Dalvik instances) require access to a valid `/data` partition. None of the services in the core class have this requirement. With only core framework services running, we unmount the `/data` volume. We then issue the `ioctl` command to delete the `dm-crypt` mapping which calls `dm-crypt`'s `crypt_dtr` destructor function [15]. The `crypt_dtr` function in turn calls `kzfree` (an atomic function to zero then free kernel memory) on the `crypt_config` structure. The FDE key (and all related key material *e.g.,* round key schedules) are in the `crypt_config` structure, and hence are securely erased before the memory is deallocated.

Vold maintains a copy of the unencrypted FDE key to speed up the password change routine (*i.e.,* changing the password will only require one invocation of PBKDF2 instead of two). We zero the memory containing vold's copy of the FDE key as well, using the `memset` function (see Section 4.2). We do the same to securely deallocate the password used to derive the KEK in the vold functions that require a password. We then mount a 128 MB tmpfs RAM disk on `/data` and set up the default directory structure that is required by the services in the main and late start classes. Finally, we set the `vold.decrypt` property to `trigger_restart_framework` which instructs init to restart the main and late start services. Upon successful framework restart, we create and set the `cryptlock.lockstate` property to 1, indicating that the device is currently running in Deadbolt mode.

While much of the Android framework gets shut down and reloaded, the Linux kernel continues running. The user will not have an interactive GUI for a brief time (see Section 4) while the tmpfs environment starts up. This can be compared to switching runlevels in a Linux OS (*e.g.,* changing from runlevel 5: full GUI to runlevel 1: single user mode). It is,



(a) Enter Deadbolt; The user may optionally choose to import some private data into Deadbolt mode, or enable *safe-mode* in which no private data may be imported or exported from Deadbolt mode.



(b) Exit Deadbolt; The user enters their FDE password to disable Deadbolt. The encrypted storage is then unlocked and remounted. The user can optionally export/merge data created in Deadbolt with the FDE mode (assuming *safe-mode* was not enabled).

Figure 2: Deadbolt Android GUI app.

however, much faster than a full reboot of the device (see Section 4).

The *cryptfs unlock* function takes (as its only parameter) the user's password to derive the KEK. The framework shutdown is triggered and then the tmpfs volume is unmounted from `/data`. The Android FDE footer is fetched, and the KEK is used to decrypt the master volume key contained within. A `dm-crypt` target is mapped from the encrypted storage volume and mounted on `/data`. We then trigger a full framework restart. Finally we set the `cryptlock.lockstate` property to 0, indicating that the device is now running in the default FDE mode. The whole process takes significantly less time that a full bootstrap from a powered off state (see Section 4). Both the lock and unlock functions can be accessed from a privileged shell through the `vdc` tool (*e.g.,* using `adb`), or by sending the vold command listener an intent from a system-privileged app with the `CRYPT_KEEPER` permission.

Deadbolt could have alternatively been implemented as two separate readable and writable `userdata` partitions, each used for varying levels of security. While this setup would avoid the need to copy data to/from Deadbolt, it would increase cognitive load on the user. With two `userdata` partitions, users would need to keep track of where certain data was created (*e.g.,* trying to recall which environment contains an SMS sent on Tuesday). We expect users to be familiar with the private session capabilities (available for some time) in modern browsers like Chrome and Firefox, so the discardable environment seemed best suited to user acceptance.

User interface and experience. We also created an Android GUI app to invoke the underlying Deadbolt logic. The app is installed in the `/system` partition, and therefore has the necessary privileges to call the vold functions. In a future version of Deadbolt, we may add the app to the power-button long-press menu, or optionally automatically activate Deadbolt after a period of device inactivity or based on the user's behavior (*e.g.,* location, time of day *etc.*; *cf.* [25]).

When the user wishes to suspend their FDE environment and enter Deadbolt mode (see step 2 in Figure 1), they launch the app. The app will check the `cryptlock.lockstate` property, to determine in which mode the device is currently running. If the device is currently in the default FDE mode, the user will be shown the *enable Deadbolt* screen (see Figure 2a). At this point the user can decide if they wish to enable Deadbolt in the safe-mode. If they do not select safe-mode, they will instead initiate the incognito-mode (see Section 3.2). The user may also choose to import certain private data from the encrypted `userdata` volume into the tmpfs environment, if incognito-mode was selected (see Section 3.4). If the user selects the safe-mode option, the `cryptlock.lockstate` property is set to 2. This informs the Deadbolt app that the options for importing/exporting private data to the tmpfs environment are disabled.

To protect any imported data and the basic functionalities of the device, a Deadbolt-specific PIN can be configured (*e.g.,* to prevent outgoing phone calls and SMS). This has the added benefit of separating the encryption password from the screen-unlock password (*i.e.,* if the adversary can guess the Deadbolt PIN, he does not gain an advantage on guessing the encryption password). In a future version of Deadbolt, we may allow the user to set this PIN before enabling Deadbolt, to save the user from the hassle of going through the system settings menu.

While the device is restarting the framework and remounting the `/data` volume, the user will not have access to the Android GUI nor any apps. They will be shown a bootup splash screen until the vold functionality has completed. This nonfunctional period is brief in comparison with a full shutdown or reboot (see Section 4).

When the device resumes in the tmpfs mode, the user will experience what is essentially a clean Android install. All settings will be reverted to defaults, and none of their after-market apps and data will be accessible. The cellular data connection is configured by selecting the appropriate data access point name (APN) entry from a pre-configured list (or optionally retrieved from a SIM card). Thus, when entering Deadbolt mode, the device should require no additional user configuration to make use of the cellular network for telephony, SMS, and mobile Internet. The user is able to perform most smart-device tasks (*e.g.,* web browsing, texting, satellite navigation) without access to the encrypted storage.

When the user wishes to exit the tmpfs environment (see step 3 in Figure 1), they will launch the Deadbolt app again. The app detects, based on the `cryptlock.lockstate` property, that the device is currently in Deadbolt mode and presents the user with the *disable Deadbolt* screen (see Figure 2b). The user then enters their password and, if safe-mode is not enabled, chooses which data should be merged back into the FDE environment. The device will again become unresponsive for several seconds before returning the user to their familiar FDE environment.

3.4 Data Transfer Between Environments

The default user experience when enabling Deadbolt is similar to that of setting up a new device: no entries exist in the address book, default wallpapers and notifications are used, wireless network passwords are undefined, *etc.* While this bare environment is sufficient for receiving phone calls and browsing the web via a data connection, users may wish to have some of their private data (from FDE mode) available in Deadbolt mode. Likewise, users may wish to export some of the data created in Deadbolt mode back to FDE mode (*e.g.,* SMS messages received while in Deadbolt). We provide functionality to import/export data between environments when Deadbolt is initialized in incognito-mode.

Importing data into Deadbolt. Importing entire content providers (*e.g.,* SQLite databases) and files from the FDE mode into Deadbolt mode is achieved by mounting the tmpfs to a temporary mount point before unmounting the encrypted `userdata` partition. After creating the default directory layout in the tmpfs, the desired files are copied from the encrypted storage to the tmpfs. The encrypted volume is then unmounted, and the tmpfs is moved (with the `MS_MOVE` mount flag) to the `/data` mount point before resuming the framework.

Although the cellular data connection is automatically configured when entering Deadbolt mode, enabling a Wi-Fi connection may be preferable for some users. As of Android 4.2, all open and pre-shared key (PSK) Wi-Fi networks and passwords are stored in the plaintext file `/data/misc/wifi/wpa_supplicant.conf`. Deadbolt can optionally (at the user's request; see Figure 2a) copy this file into the temporary environment, allowing the user to connect to access points that he/she has previously connected to.

By default, the user's address book is encrypted and locked, and therefore not available in Deadbolt. While having an empty address book does not limit the device's ability to *receive* phone calls and messages, users may find it difficult to *make* phone calls without knowing contact's numbers. Thus, it may be useful to copy the address book contacts into the Deadbolt environment. Our current prototype supports copying the address book into Deadbolt, but we note that if the device is compromised while in this mode, any contacts copied from FDE mode or created during Deadbolt mode may be accessible to an adversary if physical access to the device is gained.

Exporting data from Deadbolt. All data created while running Deadbolt (*e.g.,* snapped photos, call logs, browsing history) is written to RAM. As such, when the user switches back to the encrypted environment, this data will be overwritten by RAM assigned to the FDE mode (or optionally zeroed when exiting Deadbolt; see Section 4.2). Currently, Deadbolt supports optionally exporting all camera photos and merging call history and SMS/MMS messages into the user's encrypted storage (at the user's request; see Figure 2b).

Merging the tmpfs data into the encrypted storage before exiting Deadbolt mode requires having both volumes mounted simultaneously. The camera photos and MMS multimedia attachments are stored as files and directories. They are simply copied from the tmpfs to the encrypted storage before unmounting the tmpfs and moving the encrypted storage to the to the `/data` mount point. To merge database content, the tmpfs content provider entries are dumped and inserted into the FDE content provider databases. *e.g., sqlite3 /tmpfs/tmp_database.db .dump | sqlite3 /FDE/fde_database.db*

4. EVALUATION AND PERFORMANCE

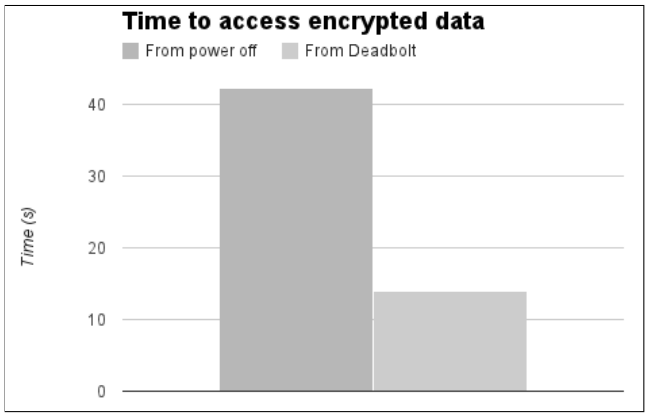
This section evaluates the performance and overhead incurred by switching to and from the Deadbolt mode. We also discuss tests performed to verify that the claimed security properties of locked FDE mode were achieved.

4.1 Performance

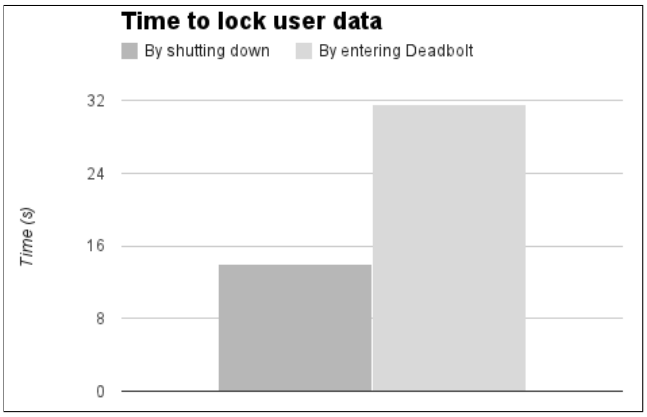
Experimental setup. All development and measurements were performed on an Asus Nexus 7 tablet running a quad-core 1.2 GHz Tegra3 processor with 1 GB of RAM. The tablet was loaded with AOSP 4.2.2 which included our vold modifications to support Deadbolt. Due to the way our system pauses and re-launches the Android framework (see Section 3.1) we are unable to rely solely on Android logs to obtain timing information. Thus, we configured the running AOSP image to save kernel messages to a log file on the internal flash filesystem. Android specific (`adb`) logs were redirected to the kernel logging facility, resulting in a unified log view and homogeneous timing data with microsecond resolution.

As a baseline, we measured the time taken to complete two standard FDE activities, as noted below.

Power-off to unlocked FDE. This test measured the time taken to arrive to an unlocked FDE state (*i.e.,* where user data is available and GUI is responsive) from a fully powered-off state (*i.e.,* where data is securely encrypted). Since we were unable to reliably measure power-on self-test (POST) time, we start our measurements at the time of logging the first Linux kernel message which typically displays the kernel version and CPU architecture. The time to input the FDE



(a) Average time to access encrypted data from power off and from Deadbolt.



(b) Average time to lock user data by shutting down and by entering Deadbolt.

Figure 3: Timing comparison for locking/unlocking encrypted storage.

password was subtracted from our total.[5] The mean over 10 trials, for arriving to an unlocked FDE state, was 42.17 (std. dev. 0.638) seconds.

Unlocked FDE to power-off. This test measured the time taken to shut down a device that is already in unlocked FDE mode. The measurements start when the power off menu item is tapped (*i.e.,* shutdown command is issued), and end at the last logged kernel message. Over 10 trials, it took an average of 14.03 (std. dev. 0.145) seconds, to go from power button pressed to full device shut down.

Next, we performed measurements with Deadbolt enabled. For these trials we measured the time taken to transition from between FDE mode to Deadbolt mode, and then return to FDE mode.

FDE to Deadbolt. Over 10 trials, the time to enable Deadbolt from unlocked FDE mode averaged 31.62 (std. dev. 1.235) seconds. While this is slightly more than double the time it takes to lock the `userdata` partition by shutting down (see Figure 3b), the user doesn't need to interact with the device once the *enable Deadbolt* sequence has been selected. Users could, if desired, start Deadbolt and immediately put their device away. The increased time is due to the initialization of system apps which must be unpacked and installed on the tmpfs volume before first use.

Deadbolt to FDE. Returning to FDE from Deadbolt took 14.00 (std. dev. 0.122) seconds on average, over 10 trials. This measurement describes the time it takes for a user to gain access to his/her encrypted data. Comparing this time to a full device power on cycle, (see Figure 3a), users would be able to unlock their data on average 28 seconds faster when using Deadbolt.

To summarize, Deadbolt allows users to more rapidly unlock private data when required by returning to FDE mode without the need for a full reboot, at the expense of longer times to lock data. However, much of the device functionality remains available in Deadbolt, helping to justify the time trade-off.

[5]We calculate the time for password input by noting the time it takes to arrive at the password prompt and subtracting the time when the password is submitted to vold.

4.2 Security Analysis

Encryption key deletion. The `dm-crypt` subsystem uses the kernel crypto module to perform AES encryption. The FDE key, along with other key material (*e.g.,* round key schedule), are present in kernel memory while the `dm-crypt` target is actively mapped. Furthermore, vold keeps a copy of the decrypted FDE key in memory after successfully mounting the `userdata` partition.

As discussed (see Section 3.3), `dm-crypt` uses the `kzfree` function to zero all key material before deallocation from RAM, when a target is unmapped [15]. Unlike the remanence properties of magnetic storage, recovering the previous state of a DRAM cell is non-trivial (*e.g.,* requiring voltage comparisons shortly after the cell has been overwritten [11]). As an additional sanity check, using the LiME memory extractor kernel module [31], we take full RAM dumps of our development device before, during, and after using Deadbolt. Statistical techniques for locating encryption keys in large amounts of data exist (*e.g.,* [28]). We make use of the AESKeyFinder [13] tool described in the original cold boot attack publication [12], to search the RAM dumps. While the FDE key was successfully located in RAM dumps from before and after using Deadbolt, no key material was located in the Deadbolt mode RAM dump.

To securely deallocate (by zeroing before freeing) vold's copy of the FDE key, we use the `memset` function, and disable compiler optimization (*i.e.,* the -O0 `gcc` flag). We also take steps to securely deallocate the password used to derive the KEK in the vold functions that require a password. When examining the region of memory that holds vold's copy of the key, we observe that the memory in fact contains zeros as opposed to the FDE key. The same is true for the password.

Physical RAM access. Despite ensuring that the key has been removed from RAM (effectively securing the encrypted data), plaintext fragments of open files from FDE mode may persist in RAM into Deadbolt mode. Android does not use swap space, so there is no danger that these fragments will be paged to disk. However, an adversary capable of obtaining a RAM dump may be able to locate plaintext private data. This problem can be mitigated by encrypting RAM, however such techniques do incur some performance overhead [23].

Other approaches include modifying the kernel, C library, and Java garbage collector memory deallocation routines to enforce memory zeroing before deallocation [32, 6]. Although these techniques also incur some performance overhead, they help ensure that any other memory containing the key or password are zeroed (*e.g.,* password entry via the Android keyboard).

We experimented with the `smem` [33] tool in an attempt to zero all available RAM before initiating the tmpfs. In its single-overwrite mode, `smem` will repeatedly call `calloc`, to allocate and zero as much heap memory as possible. However, due to the lack of swap space on Android devices, the tool was always terminated by the Linux kernel out of memory (OOM) killer before completion. On average we were able to wipe 725 MB of RAM before the process ended. We also attempted to exempt the `smem` process from the OOM killer, however that resulted in other critical processes (*e.g.,* vold) being killed instead. Disabling the OOM killer entirely (by setting `/proc/sys/vm/overcommit_memory` to 2), generally resulted in a kernel panic or system hang. To protect against plaintext recovery from RAM dumps, Deadbolt should be used with a kernel based secure page deallocator. Grsecurity's PaX extensions provide the capability to wipe all memory pages as soon as they are freed with minimal overhead (roughly 3% performance impact during the task of kernel compilation[6]). All Android framework and user apps are killed when Deadbolt is enabled, hence all private data should be erased from RAM.

When disabling the Deadbolt environment, plaintext fragments from logs and activities in the tmpfs may be retrieved from a memory dump in FDE mode. Since the tmpfs is a finite size, we were able to successfully wipe the tmpfs using Android's `dd` tool, before unmounting and returning to FDE mode. On average, wiping the 128 MB tmpfs required 2.4 seconds.

Security of imported data. If the user has not set a Deadbolt PIN (see Section 3.3) data imported into Deadbolt may be trivially retrieved by an adversary with access to the device. An adversary may obtain a list of imported Wi-Fi access points/passwords as well as contacts *etc.* if the device is stolen (See Section 3.4).

In summary Deadbolt protects against theft of data that is encrypted and locked, but offers no additional protection for data available to the user and system while in Deadbolt mode. This data may be susceptible to compromise through malware or physical device theft. Deadbolt does not protect against off-line password guessing attacks, so a strong FDE password is still recommended.

5. DISCUSSION

This section describes the trade-offs of including certain enhancements in Deadbolt. We then discuss limitations of the system as implemented.

5.1 Adding functionality to Deadbolt

In its default configuration, Deadbolt includes a base set of applications providing basic smart-device functionality. Users can optionally access certain types of data (*e.g.,* contacts and Wi-Fi passwords; see Section 3.4), but they may wish to have other types of data accessible within the Deadbolt environment.

Applications with Notifications. Mobile devices today are often used to receive alerts of various types (*e.g.,* email, tweets, social network messages, *etc.*). Since most such notification-enabled apps are not launched while the device is in Deadbolt mode, users may need to configure notifications over SMS if supported by the service. Alternatively, installing the app in Deadbolt and entering user credentials (see below) is possible, but this is inconvenient, and the credentials and received data will remain unencrypted and at risk. To limit exposure, some providers offer finer granularity in access to cloud data. For example, Google's application-specific passwords[7] could allow an application to retrieve unread email counts, but not sign in to Gmail.

Other Apps and User Data. Users may require additional apps beyond those included in the base Android install (see Section 2.2) while in Deadbolt mode (*e.g.,* a secure SMS application or a different browser). We are considering adding a feature where users can optionally specify which user-installed apps should be copied from FDE mode and made available in Deadbolt mode. This can be achieved by copying the apk package file from the FDE volume to the tmpfs volume, and invoking the package manager to install the app after the Deadbolt mode has been initialized. To address the lack of a uniform storage location or format (*e.g.,* settings databases, flat files, content providers), a Deadbolt-specific data storage API could be implemented. This API would allow developers to store specific, non-private data that could be used safely within Deadbolt.

5.2 Comparison to Existing Encryption Techniques

This section compares the security properties of Deadbolt to other existing encryption techniques (see Section 2.1). Table 1 provides a non-exhaustive list of vendor solutions to data encryption (separated by either per-file encryption or full disk encryption). For each column, we assume the device has been configured to enable secure data storage (*e.g.,* turning on full disk encryption or file based encryption) as well as a screen lock. Columns 1 and 2 (resilience to lock-screen bypass and cold boot attacks, respectively) assume the device data is secure even if obtained by an adversary while powered-on but locked. Column 3 refers to the need to include an additional hardware component on the device, or, alternatively, to modify (*e.g.,* burn in cryptographic material) the device at manufacturing time. The *app notifications* column describes the ability for a device in powered-on but screen-locked mode to receive and display app notifications to the user.

As Table 1 shows, Deadbolt offers the security benefits of using file-based encryption without the need to use specialized hardware or drastically re-engineer Android's full-disk encryption approach. While Deadbolt loses the generic ability to receive notifications for all third party applications, it gains the advantage of offering an incognito mode (see Section 3.1).

5.3 Limitations

[6] `http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Sanitize_all_freed_memory`

[7] `https://support.google.com/accounts/answer/185833?hl=en`

		Lock-screen bypass resilient	Cold-boot resilient	Software only	App notifications	Incognito mode
File	Apple iOS [2]		•		•	
	Blackberry [24]	•	•	•	•	
FDE	Windows Phone [17]				•	
	Android FDE [1]			•	•	
	Deadbolt	•	•	•	○[a]	•

Table 1: Comparison of Deadbolt to existing device encryption techniques.

[a]Deadbolt itself does not preclude receiving notifications (*e.g.,* over SMS), however, these types of notifications must be offered by the service provider (see Section 5.1).

This section discusses some of the limitations of Deadbolt including system requirements, and precautions that must be taken when using the system.

Usability considerations. The required time to enter the Deadbolt environment is non-negligible, and thus may discourage users from frequently enabling Deadbolt. We note, however, that devices are typically not required for use immediately after enabling the lock-screen (or Deadbolt) so we see this as only a minor inconvenience. The absence of some user applications and app notifications is a more significant usability challenge. App notifications through SMS (which are available from some service providers; *e.g.,* Twitter) is a viable solution. Alternatively, services may provide a privacy preserving meta-notification without requiring the user's login password (*e.g.,* notifying the user that unread emails exist, without disclosing contents). To alleviate the user from the burden of manually enabling Deadbolt, automatic policies may be enabled to activate Deadbolt after a period of device inactivity or based on the user's behavior (*e.g.,* location, time of day *etc.*; *cf.* [25]).

Root access. Deadbolt requires a modified `vold` binary, which exists in the system partition. On Android, root access is required to modify files on the system partition, and therefore is also required to install Deadbolt. Patches to vold will be submitted for inclusion in AOSP, which in the future may allow users to install Deadbolt by way of a userspace app download.

Memory requirements. RAM allocated to temporary filesystems is unavailable for use as regular memory for applications. Thus, it is important to find the right balance between performance and functionality such that there is enough reserved space for user created content, but also enough RAM to prevent Android from killing processes and slowing down the system.

On first use, system apps (*i.e.,* Android's base set of applications) are unpacked into the `/data` partition. We observed that for AOSP 4.2.2, `/data` required a minimum of 44-81 MB (depending on tablet or phone). Mounting this partition in RAM and allocating memory for apps necessitates at least 128 MB of RAM.[8] This memory requirement may preclude lower-end devices with less than 256 MB of RAM from running Deadbolt. However, Deadbolt works with the Android storage encryption mechanism which was made available for both smartphones and tablets in version 4.0. According to the Android 4.0 Compatibility Definition Document (CDD), all 4.0 capable devices must have at least 340 MB of memory available to the kernel and userspace.[9] Thus Deadbolt is compatible with any device that supports encryption. During testing, we found a safe minimum amount of RAM is 128 MB for the tmpfs, but assigning half of available RAM to the tmpfs provides balance between performance and available space.

6. RELATED WORK

Several academic proposals exist to protect encrypted data at rest from attack. In general, related research falls under three main categories which we discuss in this section.

Cold boot attacks and defenses. The cold boot attack [12] aims to retrieve encryption keys from RAM when security mechanisms such as lock-screens are in use, by exploiting data remanence properties of DRAM. Unlike magnetic and Flash storage, DRAM is not persistent. Nevertheless, it will retain data for a short time after power has been removed [11]. The cold boot attack is executed by rebooting the machine into a custom bootloader or OS to retrieve encryption keys from memory (or by transplanting the RAM into a custom device to do the same). DRAM has better remanence properties at lower temperatures, so the attack often involves freezing the physical memory [12]. The cold boot attack has been demonstrated for Android's FDE implementation to successfully capture the FDE key from RAM [20].

Defenses against the cold boot attack often rely on keeping encryption keys, and intermediate state, outside of RAM (*e.g.,* in CPU cache, registers [19, 18, 29], or external devices [5, 32]). This technique has recently been demonstrated for ARM based Android devices by Götzfried *et al.* [10]. These methods tend to have non-negligible performance impacts, as the CPU registers or cache must be reserved for the encryption key and other processes will not have access to them. To minimize the performance impact, these techniques could be employed only while the screen is locked. However, even if such defenses are in place, data is only as secure as the lock-screen, since the encrypted volume is still unlocked. In that respect, Deadbolt could be combined with such a technique to provide additional protection when needed.

Deadbolt does not attempt to maintain key material outside of RAM, but rather ensures the `userdata` partition is locked and the keys are securely deallocated when the device enters this state. This gives a device in Deadbolt mode the security properties of being powered off, while retaining some of the usability benefits of being powered on.

DMA attacks and defenses. Direct memory access (DMA) techniques can be used to access physical RAM and attempt to search for cryptographic keys [26]. Furthermore, DMA attacks can be used to manipulate a running kernel to read

[8]Strictly speaking, only 82 MB of RAM are required, but users would run into "out of space" messages as soon as a few photos are taken or as browser history grows.

[9]`http://source.android.com/compatibility/4.0/android-4.0-cdd.pdf`

CPU state [4]. ARM CPUs expose debug commands which can be used to access RAM and CPU state [3]. These commands can be used to compromise an ARM-based Android device with a debug interface such as JTAG [29].

Current defenses against DMA attacks tend to rely on virtual machine monitors (VMM) (see *e.g.,* [21]). Deadbolt's design goals are to avoid the need for additional hardware, and overhead required by a VMM. Our technique is not susceptible to DMA attacks, since the key material is neither in RAM nor the CPU registers. We also ensure that the encrypted volume is locked, and optionally attempt to flush plaintext fragments from RAM, when entering Deadbolt mode (see Section 4.2).

VM based protection. The use of virtual machines (VM) has also been suggested for performing privacy-sensitive activities (see *e.g.,* [16]). These proposals require hardware virtualization support which (as of writing) is uncommon in mobile devices. Additionally, vulnerabilities in the hypervisor may lead to compromise of the FDE key, if it is still in RAM when the VM is running. Deadbolt re-launches the Android framework from a running kernel, instead of from within another instance (*cf.* switching Linux runlevels). This greatly reduces the time and resource overhead required for Deadbolt, as explained in Section 4.

7. CONCLUSION

The always-on mobile computing paradigm presents a new set of challenges for robust, software-only full-disk encryption. Even when FDE is enabled, data tends to remain unlocked when devices are powered-on. In this paper, we have proposed a first step towards providing strong security properties while allowing basic functionality of smart-devices to remain enabled. Our proposal, Deadbolt, allows security-conscious users to enable an additional level of security in higher threat environments, helping secure their FDE key and data. Deadbolt requires no additional hardware support, and is implemented to be compatible with the modern Android builds. For source code for the Deadbolt app and vold patches, please contact the authors.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. We also thank members of the Carleton Computer Security Lab for their enthusiastic discussion on this topic. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)—the second author through a Canada Graduate Scholarship; the third through a Discovery Grant and as Canada Research Chair in Authentication and Computer Security. We also acknowledge support from NSERC ISSNet.

8. REFERENCES

[1] Android Open Source Project. Notes on the implementation of encryption in Android 3.0. Online document (Feb. 2011). http://source.android.com/tech/encryption/android_crypto_implementation.html.

[2] Apple. iOS security. Technical document (Oct. 2012). http://www.apple.com/ipad/business/docs/iOS_Security_Oct12.pdf.

[3] ARM. Cortex-A8 technical reference manual. Technical document (2007). http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf.

[4] Blass, E.-O., and Robertson, W. TRESOR-HUNT: attacking cpu-bound encryption. In *Annual Computer Security Applications Conference (ACSAC'12)* (Orlando, Florida, 2012).

[5] Chen, Y., and Ku, W.-S. Self-encryption scheme for data security in mobile devices. In *Consumer Communications and Networking Conference, (CCNC 2009)* (Las Vegas, NV, 2009).

[6] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding your garbage: reducing data lifetime through secure deallocation. In *USENIX Security Symposium* (Baltimore, MD, 2005).

[7] DMCrypt. dm-crypt: Linux kernel device mapper crypto target. Online document (July 2012). https://code.google.com/p/cryptsetup/wiki/DMCrypt.

[8] Dunn, A. M., Lee, M. Z., Jana, S., Kim, S., Silberstein, M., Xu, Y., Shmatikov, V., and Witchel, E. Eternal sunshine of the spotless machine: protecting privacy with ephemeral channels. In *USENIX Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, 2012).

[9] Ferguson, N. AES-CBC + Elephant diffuser: A Disk Encryption Algorithm for Windows Vista. Technical document (Aug. 2006). http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/bitlockercipher200608.pdf.

[10] Götzfried, J., and Müller, T. ARMORED: CPU-bound encryption for Android-driven ARM devices. In *Conference on Availability, Reliability and Security (ARES 2013)* (Regensburg, Germany, 2013).

[11] Gutmann, P. Data remanence in semiconductor devices. In *USENIX Security Symposium* (Washington, D.C., 2001).

[12] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium* (San Jose, CA, 2008).

[13] Heninger, N., and Feldman, A. J. AESKeyFinder, 2008. Version 1.0 (Jul. 2008). https://citp.princeton.edu/research/memory/code/.

[14] Kaliski, B. PKCS #5: Password-based cryptography specification, version 2.0, Sept. 2000. RFC 2898 (informational).

[15] Linux Kernel. dm-crypt.c: Linux kernel source. Source Code (July 2013). https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/md/dm-crypt.c.

[16] Mannan, M., Kim, B. H., Ganjali, A., and Lie, D. Unicorn: two-factor attestation for data security. In *ACM Computer and Communications Security (CCS'11)* (Chicago, IL, 2011).

[17] Microsoft. Windows Phone 8 security overview v1.1. Technical document (Dec. 2012). http://go.microsoft.com/fwlink/?LinkId=266838.

[18] Müller, T., Dewald, A., and Freiling, F. C. AESSE: a cold-boot resistant implementation of AES. In *Proceedings of the Third European Workshop on System Security (EUROSEC'10)* (Paris, France, 2010).

[19] Müller, T., Freiling, F. C., and Dewald, A. TRESOR runs encryption securely outside RAM. In *USENIX Security Symposium* (San Francisco, CA, 2011).

[20] Müller, T., and Spreitzenbarth, M. FROST: Forensic recovery of scrambled telephones. In *Applied Cryptography and Network Security (ACNS'13)* (Banff, Canada, 2013).

[21] Müller, T., Taubmann, B., and Freiling, F. TreVisor: OS-independent software-based full disk encryption secure against main memory attacks. In *Applied Cryptography and Network Security(ACNS'12)* (Singapore, 2012).

[22] Ooi, T. Yet another Android rootkit – /protecting/system/is/not/enough/. In *Black Hat* (Abu Dhabi, UAE, 2011).

[23] Peterson, P. Cryptkeeper: Improving security with encrypted RAM. In *Technologies for Homeland Security (HST 2010)* (Waltham, MA, 2010).

[24] RIM. Blackberry enterprise server 5.0.2–security technical overview. Technical document (Mar. 2011). http://docs.blackberry.com/en/admin/deliverables/16648/.

[25] Riva, O., Qin, C., Strauss, K., and Lymberopoulos, D. Progressive authentication: deciding when to authenticate on mobile phones. In *USENIX Security Symposium* (Bellevue, WA, 2012).

[26] Sang, F., Nicomette, V., and Deswarte, Y. I/O attacks in Intel PC-based architectures and countermeasures. In *SysSec Workshop (SysSec 2011)* (Amsterdam, The Netherlands, 2011).

[27] Schaad, J., and Housley, R. Advanced Encryption Standard (AES) key wrap algorithm, Sept. 2002. IETF RFC 3394.

[28] Shamir, A., and Someren, N. v. Playing "hide and seek" with stored keys. In *Financial Cryptography (FC'99)* (Anguilla, BWI, 1999).

[29] Simmons, P. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Annual Computer Security Applications Conference (ACSAC'11)* (Orlando, Florida, 2011).

[30] Skillen, A., and Mannan, M. On implementing deniable storage encryption for mobile devices. In *Network and Distributed System Security Symposium (NDSS 2013)* (San Diego, CA, Feb. 2013).

[31] Sylve, J. LiME - Linux memory extractor source code, 2013. Presented at ShmooCon'12. Version 1.1 (Mar. 2013). https://code.google.com/p/lime-forensics/.

[32] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. CleanOS: limiting mobile data exposure with idle eviction. In *USENIX Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, 2012).

[33] THC. Secure-delete toolkit, 2003. Version 3.1 (Nov. 2003). http://www.thc.org/releases.php?q=delete.

[34] Wadner, K. iOS security. Technical document (July 2011). http://www.sans.org/reading_room/whitepapers/pda/security-implications-ios_33724.