Implementing and Optimizing an Encryption Filesystem on Android

Zhaohui Wang, Rahul Murmuria, Angelos Stavrou
Department of Computer Science
George Mason University
Fairfax, VA 22030, USA
zwange@gmu.edu, rmurmuri@gmu.edu, astavrou@gmu.edu

***Abstract*—The recent surge in popularity of smart handheld devices, including smart-phones and tablets, has given rise to new challenges in protection of Personal Identifiable Information (PII). Indeed, modern mobile devices store PII for applications that span from email to SMS and from social media to location-based services increasing the concerns of the end user's privacy. Therefore, there is a clear need and expectation for PII data to be protected in the case of loss, theft, or capture of the portable device.**

In this paper, we present a novel FUSE (Filesystem in USErspace) encryption filesystem to protect the removable and persistent storage on heterogeneous smart gadget devices running the Android platform. The proposed filesystem leverages NIST certified cryptographic algorithms to encrypt the data-at-rest. We present an analysis of the security and performance trade-offs in a wide-range of usage and load scenarios. Using existing known micro benchmarks in devices using encryption without any optimization, we show that encrypted operations can incur negligible overhead for read operations and up to twenty (20) times overhead for write operations for I/O-intensive programs. In addition, we quantified the database transaction performance and we observed a 50% operation time slowdown on average when using encryption. We further explore generic and device specific optimizations and gain 10% to 60% performance for different operations reducing the initial cost of encryption. Finally, we show that our approach is easy to install and configure across all Android platforms including mobile phones, tablets, and small notebooks without any user perceivable delay for most of the regular Android applications.

***Keywords*-Smart handheld devices, Full disk encryption, Encrypted filesystem, I/O performance.**

I. Introduction

Technology trends in both hardware and software have driven the hardware industry towards smaller, faster and more capable mobile hand-held devices that can support a wider-range of functionality and open source operating systems. Mobile hand-held devices are popularly called smart gadgets(e.g. smartphones, tablets, e-book readers). The smart gadget life-cycle has evolved drastically in recent years. Nielsen market data trends [9] shows that mobile devices have lifetimes of approximately 6 months between generations. Numerous factors influenced the industry to grow at this fast pace. One of the most important reasons was the availability of operating systems for mobile handheld devices that were hardware-agnostic by design.

These new generations of the smart gadget devices such as the iPhone and Google Android devices are powerful enough to accomplish most of the tasks that previously required a personal computer. Indeed, this newly acquired computing power gave a rise to plethora of applications that attempt to leverage the new hardware. These include but are not limited to Internet browsing, email, messaging, social networking, and GPS navigation.

However, smart gadgets have to come a long way in terms of security. Organizations have come to realize that these commercially available smart gadgets will soon have to serve as an integral part of their operations. This requires a level of security that allows for security of data at-rest and on the move to support secure communications. A major obstacle is that there is a serious lack of National Institute of Standards (NIST) approved encryption algorithms on these commercially available smart gadgets. Much less common is the existence of any encryption techniques that can pass the strong government validation process in place for any computing device to be used in an adversarial environment. Also, the expectation for each individual application to support encryption runs into the key management problem: other applications in the system can potentially gain access to the key and render the encryption useless. Therefore, there is a need for a practical approach to build common security libraries that operate at the operating system level and provide strong encryption. This system has to be ubiquitous and integrate into the ecosystem of smart gadgets with minimal maintenance and installation cost.

Encryption however comes at a significant performance cost. On smart gadgets where resources, like the battery, are very limited, it is important to keep a low footprint on such solutions. In this paper, we focus on analysing the performance for persistent storage protection using encryption on smart gadget devices. We use a filesystem encryption which uses certified cryptographic algorithms to store encrypted versions of every file in a source directory. The volume key is decrypted using a password supplied by the user. This is different from full-disk encryption software because the protected data is mount in memory at a specified mount point in the filesystem. Moreover, the features and restrictions depend on the underlying partition's filesystem type.

A detailed comparison of a filesystem-encryption over a kernel-based full-disk encryption is beyond the scope of this paper. However, we list some of the advantages that applied to us.

- We were able to leverage NIST validated cryptographic libraries [6] which are not implemented in kernel-space.
- Our implementation can be extended to different hardware with negligible effort.
- We focus on data encryption without having to deal with other aspects of filesystem design. Underlying filesystems like *ext3* and *yaffs2* already have strong support for handling data-corruption and journalling.

Since file I/O operations on the mount point eventually hit the encrypted copy of the file on the underlying filesystem, various performance optimizations can be possible by adjusting the filesystem parameters such as block size, buffer size. In addition, we analyse the performance of various SQLite database transactions on Android.

The main contributions of this paper are summarized as follows:

- We are the first to study the filesystem encryption's performance on commodity smart gadget devices given modern NAND technology as storage media. To that end, we ported an open-source cryptographic filesystem, EncFS, on commodity Android systems
- We present benchmark results of the EncFS running on Android system with various I/O operations. Particularly, we focus our analysis on the security versus performance trade-off including SQLite database transactions
- Finally, we discuss the limitations of filesystem encryption and demonstrate that it is feasible on smart gadget devices with a reasonable performance overhead.

The rest of this paper is organized as follows: Section II presents related research on mobile operating system and filesystem benchmarking. Section III introduces the background information and our threat model. The design and implementation of EncFS on Android platform fits in Section IV. In Section V, we discuss the performance results under different filesystem operations and offer optimization solutions and Section VII concludes the paper.

II. Related works

Secure Storage: As the amount of stored digital data follows an explosive trend, so does the theft of sensitive data through the loss or misplacement of laptops, smart gadget devices, flash drives, portable hard drives, and other electronic storage media. Sensitive data may also be leaked accidentally due to improper disposal or resale of storage media. Diesburg et al. [17] surveys, summarizes and compares existing methods of providing confidential storage and deletion of data in personal computing environments given that we must protect the secrecy of the entire data lifetime by employing confidential ways to store and delete data. Lee et al. [23] designed a NAND flash file system with a secure deletion functionality by using encryption to delete files and forces all keys of a specific file to be stored in the same block. Keypad [20] presents an auditing file system for theft-prone devices, such as laptops and smart gadget devices. Keypad supports fine-grained file auditing and can disable future file access after a device's loss, even in the absence of device network connectivity.

There are existing full disk encryption solutions for Android device[1], [13]. Both techniques leverage Linux in-kernel crypto APIs and insert an additional device-mapper layer to achieve transparent full disk encryption. Android encryption [1] uses 128bits AES with CBC and ESSIV:SHA256. However, there is no systematic benchmark for either of them yet.

Filesystem Performance: Kim et al. [22] studied the performance of web browsing and application installation under various storage device and concludes the performance variation can be attributed to the characteristics of the storage device, the workload pattern (random or sequential), and the operating system.

Existing research work on FUSE-based filesystem benchmarks only focus on traditional desktop system and comparison to other in-kernel filesystems [30]. To the best of our knowledge, current mobile database research focus on universal accessibility [31], [32] and there is much research space on FUSE-based file-system benchmarking on modern smart gadget devices.

Mobile OS Attacks and Defenses: The emerging threats brought by smart gadget devices and defense approaches are also well studied by the research community. The presentation "Understanding Android's Security Framework" [19] presents a high-level overview of the mechanisms required to develop secure applications within the Android development framework. The tutorial contains the basics of building an Android application. However, the described interfaces must be carefully secured to defend against general malfeasance. They showed how Android's security model aims to provide mechanisms for requisite protection of applications and critical smart phone functionality and present a number of best practices for secure application development within the environment. However, authors in [28] showed that this is not enough and that new semantically rich and application-centric policies have to be defined and enforced for Android. Moreover, in [26] the authors show how to establish trust and measure the integrity of application on mobile phone systems.

Racic and Kim et al. [29], [21] studied malware that aims to deplete the power resources on the mobile devices. The provided solutions involve changes in the GSM telephony infrastructure. Their work shows that attacks were mainly carried out through the MMS/SMS interfaces on the device. In addition, in [25] the authors show that applications can simply overuse the WiFi, Bluetooth or display of the device

Table I
GOOGLE NEXUS S HARDWARE MODULES.

Modules	Hardware
CPU	Samsung Intrinsity S5PV210 1Ghz
GPU	PowerVR SGX 540
Mother board	Samsung S3C SoC
RAM	512 MB, 345MB Application Processor accessible
ROM	1981 MB , partitioned as boot/system/userdata/cache and radio
External Storage	13.3GB VFAT partition
Camera	5 MegaPixels Sensor_S5KA3DFX
Wifi+BlueTooth+FM	Boardcom BCM 4329, 802.11a/b/g/n
Touch Screen Input	Atmel MaxTouch 224
Digital Compass	AK8973 compass
Accelerometer	KR3DM sensor
Near Field Communication	NXP PN544 NFC

and eventually cause a denial of service attack. VirusMeter [24] models the power consumption and detects the malware based on power abnormality. However, the use of linear regression model with static weights for devices' relative rate of battery consumption is a non-scalable approach [27].

Bickford et al. [14] uses three example rootkits to show that smart phones are just as vulnerable to rootkits as desktop operating systems. However, the ubiquity of smart phones and the unique interfaces that they expose, such as voice, GPS and battery, make the social consequences of rootkits particularly devastating. Cloaker [16] is a non-persistent rootkit that does not alter any part of the host operating system (OS) code or data, thereby achieving immunity to all existing rootkit detection techniques which perform integrity, behavior and signature checks of the host OS. Cloaker leverages the ARM architecture design to remain hidden from current deployed rootkit detection techniques, therefore it is architecture specific but OS independent. Bojinov et al. proposed a mechanism of executable ASLR that requires no kernel modifications for defending remote code injection attacks for mobile devices [15]. TaintDroid [18] addresses the security issues with dynamic information flow and privacy on mobile handheld devices by tracking application behavior to determine when privacy-sensitive information is leaked. This includes location, phone numbers and even SIM card identifiers, and to notify users in realtime. Their findings suggest that Android, and other phone operating systems, need to do more to monitor what third-party applications are doing when running in smart phones. Our encryption filesystem protects the static data on storage in complimentary.

III. BACKGROUND & THREAT MODEL

A. Background

Google's Android is a comprehensive software framework for mobile devices (i.e., smart phones, PDAs), tablet computers and set-top-boxes. The Android operating system includes the system library files, middle-ware, and a set of standard applications for telephony, personal information management, and Internet browsing. The device resources, like the camera, GPS, radio, and Wi-Fi are all controlled through the operating system. Android kernel is based on an enhanced Linux kernel to better address the needs of mobile platforms with improvements on power management, better handling of limited system resources and a special IPC mechanism to isolate the processes. Some of the system libraries included are: a custom C standard library (Bionic), cryptographic (OpenSSL) library, and libraries for media and 2D/3D graphics. The functionality of these libraries are exposed to applications by the Android Application Framework. Many libraries are inherited from open source projects such as WebKit and SQLite. The Android runtime comprises of the Dalvik, a register-based Java virtual machine. Dalvik runs Java code compiled into a *dex* format, which is optimized for low memory footprint. Everything that runs within the Dalvik environment is considered as an application, which is written in Java. For improved performance, applications can mix native code written in the C language through Java Native Interface (JNI). Both Dalvik and native applications run within the same security environment, contained within the 'Application Sandbox'. However, native code does not benefit from the Java abstractions (type checking, automated memory management, garbage collection). Table I lists the hardware modules of Nexus S, which is a typical Google branded Android device.

Android's security model differs significantly from the traditional desktop security model [2]. Android applications are treated as mutually distrusting principals; they are isolated from each other and do not have access to each others' private data. Each application runs within their own distinct system identity (Linux user ID and group ID). Therefore, standard Linux kernel facilities for user management is leveraged for enforcing security between applications. Since the Application Sandbox is in the kernel, this security model extends to native code. For applications to use the protected device resources like the GPS, they must request for special permissions for each action in their Manifest file, which is an agreement approved during installation time.

Android has adopted SQLite [12] database to store structured data in a private database. SQLite supports standard relational database features and requires only little memory at runtime. SQLite is an Open Source database software library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. Android provides full support for SQLite databases. Any databases you create will be accessible by name to any java class in the application, but not outside the application. The Android SDK includes a *sqlite3* database tool that allows you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases. Applications written by 3rd party vendors tend to use these database features extensively in order to store data on internal memory. The databases are stored as single files in the filesystem and carry the permissions for only the application that created the file to be able to access it. Working with databases in Android, however, can be slow due to the necessary I/O.

EncFS is a FUSE-based file-system offering file-system encryption on traditional desktop operating systems. FUSE is the supportive library to implement a fully functional filesystem in a userspace program [5]. EncFS uses the FUSE library and FUSE kernel module to provide the file-system interface and runs without any special permissions. EncFS runs over an existing base file-system (for example, ext4,yaffs2,vfat) and offers the encrypted file-system. OpenSSL is integrated in EncFS for offering cryptographic primitives. Any data that is written to the encrypted file-system is encrypted transparently from the user's perspective and stored onto the base file-system. Reading operations will decrypt the data transparently from the base filesystem and then load it into memory.

B. Threat Model

Handheld devices are being manufactured all over the world and millions of devices are being sold every month to the consumer market with increasing expectation for growth and device diversity. The price for each unit ranges from free to eight hundred dollars with or without cellular services. In addition, new smartphone devices are constantly released to the market which results the precipitation of the old models within months of their launch. With the rich set of sensors integrated with these devices, the data collected and generated are extraordinarily sensitive to user's privacy. Smartphones are therefore data-centric model, where the cheap price of the hardware and the significance of the data stored on the device challenge the traditional security provisions. Due to high churn of new devices it is compelling to create new security solutions that are hardware-agnostic.

While the Application Sandbox protects application-specific data from other applications on the phone, sensitive data may be leaked accidentally due to improper placement, resale or disposal of the device and its storage media (e.g. removable sdcard). It also can be intentionally exfiltrated by malicious programs via one of the communication channels such as USB, WiFi, Bluetooth, NFC, cellular network etc. For example, an attacker can compromise a smartphone and gain full control of it by connecting another computing device to it using the USB physical link [33]. Moreover, by simply capturing the smartphones physically, adversaries have access to confidential or even classified data if the owners are the government officials or military personnels. Considering the cheap price of the hardware, the *data* on the devices are more critical and can cause devastating consequences if not well protected. To protect the secrecy of the data of its entire lifetime, we must have robust techniques to store and delete data while keeping confidentiality.

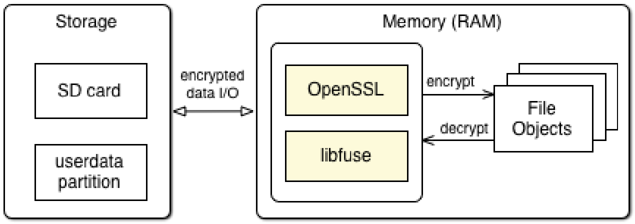

Figure 1. Abstraction of Encryption Filesystem on Android

In our threat model, we assume that an adversary is already in control of the device or the bare storage media. The memory-borne attacks and defences are out of the scope of this paper and addressed by related researches in Section II. A robust data encryption infrastructure provided by the operating system can help preserve the confidentiality of all data on the smartphone, given that the adversary cannot obtain the cryptographic key. Furthermore, by destroying the cryptographic key on the smartphone we can make the data practically irrecoverable. Having established a threat model and listed our assumptions, we detail the steps to build encryption filesystem on Android in the following sections.

IV. System Overview

A. EncFS for Android

EncFS is selected as the basis for our encryption filesystem. We introduced the filesystem in Section III-A.

Three major components are required to make EncFS work on any platform: kernel FUSE library support, user space *libfuse*, and EncFS binaries. To make an encryption file-system work on Android, a modified bootstrapping process and password login was integrated into the operating system framework.

EncFS uses standard OpenSSL cryptographic libraries in userspace. This gives us various advantages over using a kernel-based cryptographic library. Some of the features of our solution verses other in-kernel encryption approaches [1], [13] are as follows:

- By using EncFS our system is backward and forward compatible with existing and future Android versions.

Since *libfuse* and *libc* are stable across different versions of Android and different devices, only minimal engineering efforts are needed (if any) to make EncFS work on other variations of Android-based smart devices.

- EncFS leverages OpenSSL suite as the crypto engine. The OpenSSL libraries, libcrypto and libssl, implement various cryptographic algorithms that are validated and in compliance with FIPS 140-2 Level:1 standard [10].
- In addition, our approach supports all underlying file-systems, including *yaffs2*, *ext4* and *vfat*.

In order to build EncFS, we created a package with the components described below. It is required to root the phone in order to prepare it with the modified kernel, system binaries and java framework patches discussed below. Once installed, EncFS does not need processes or applications to run as root, in order to encrypt their data. The userspace will function without knowledge of any change in the underlying layers.

Kernel FUSE support: In general, FUSE module provides a bridge to the actual kernel interfaces. However, the Android Linux kernel does not support FUSE file-systems by default to eliminate redundant functionalities that are not required by Android. Most Android devices, including the Nexus S which we use, do not come with the FUSE modules enabled in the kernel. We obtain the kernel source code from Google's Android Open Source Project (AOSP) website and enabled the kernel FUSE modules necessary for *libfuse* to run. We then flash our device with this customized kernel.

Libfuse: As the fundamental supportive library for all FUSE-based file-systems, *libfuse* is not officially supported in the Android system. Furthermore, The Bionic C library in Android is missing glue layer code for interfacing VFS (Virtual FileSystem in Linux) and FUSE. We patched the Bionic C library with missing header files (*statvfs*) and corresponding data structures that are required for *libfuse* version 2.8.5.

EncFS: By building the EncFS sources for the ARM architecture, we created the executables that would enable us manage the EncFS filesystem. In addition to *libfuse*, EncFS also depends on the boost library which is a widely adopted C++ library[3], *librlog* for logging[8] and *libcrypto/libssl* for cryptographic primitives. We patched boost library version 1.45 which is the current-to-date version as of this development and built it against Android Bionic C library. The *librlog* is versioned at 1.4 while the OpenSSL suite included with Android 2.3 is 1.0.0a.

EncFS supports two block cipher algorithms: AES and Blowfish. AES runs as a 16 byte block cipher while Blowfish runs as a 8 byte block cipher. Both algorithms support key lengths of 128 to 256 bits and block sizes of 64 to 4096 bytes. Since AES is selected by US government as standard block cipher, our experiments in Section V focus on AES only.

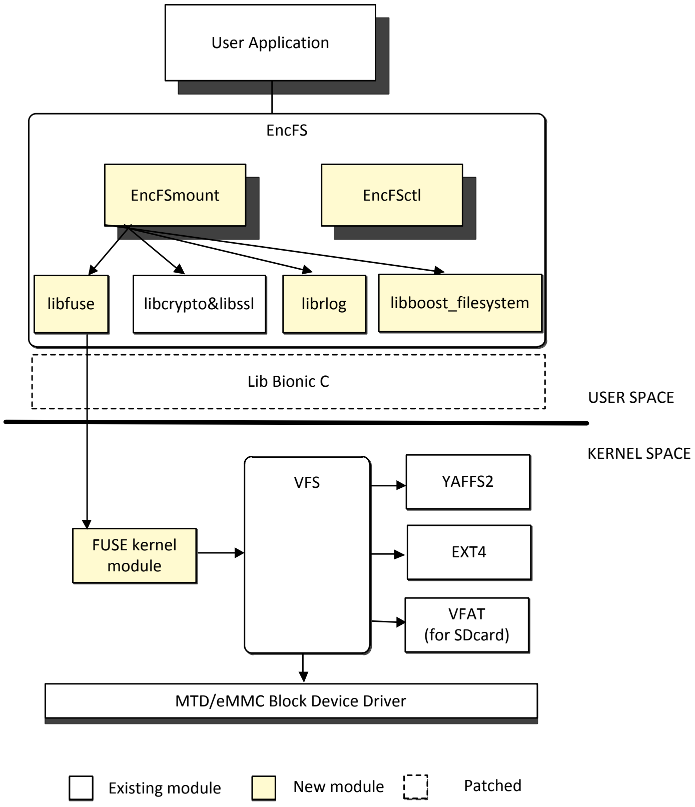

Figure 2. The EncFS Layout.

Depending on whether we built them as static or shared libraries, we push the compiled binaries onto the required locations on the phone. Figure 2 illustrate the overall layout of EncFS.

User Interface: Normally, the Android framework loads the user interface by unpacking the applications and other files from */system* and */data* partitions. The */data* partition contains all the user-installed applications and all other user-specific data. In our implementation, this */data* partition contains only a skeleton of the required folders which won't be used by the users for actual data. We store the encrypted data in a separate directory and mount it over */data* partition when the user supplies the password.

We modified the Launcher application in Android framework to accept this password, which is the key for the encrypted version of the */data* partition. If the password provided by the user is valid, EncFS mounts the encrypted data partition on */data* mountpoint using FUSE. If the mount is performed successfully, the Launcher will call a dedicated native program installed by us to *soft* reboot Android Dalvik environment and the user is presented with his encrypted userdata partition, decrypted into the memory transparently.

The user has limited number of login attempts. If the failure attempts accumulates to a predefined threshold value (10 in our case), the Launcher program will erase all the

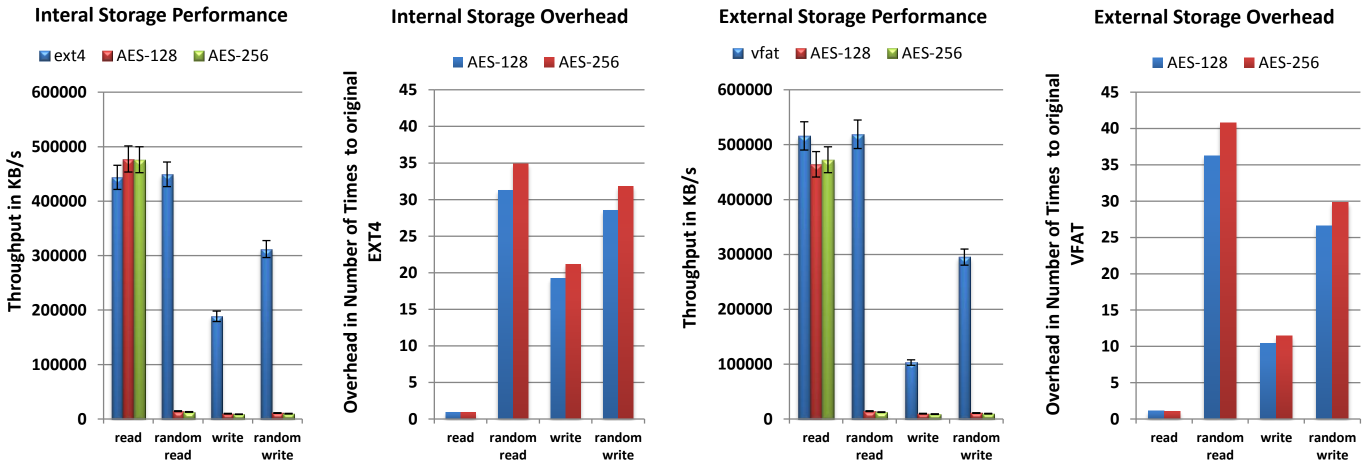

Figure 3. EncFS Basic I/O Throughput and Normalized Overhead

Table II
GOOGLE NEXUS S NAND PARTITION LAYOUT.

Dev	Size	Name
/dev/mtd/mtd0:	2MB	bootloader
/dev/mtd/mtd1:	1MB	misc
/dev/mtd/mtd2:	8MB	boot
/dev/mtd/mtd3:	8MB	recovery
/dev/mtd/mtd4:	469.5MB	cache
/dev/mtd/mtd5:	13MB	radio
/dev/mtd/mtd6:	6MB	efs
/dev/block/mmcblk0p1:	512MB	system
/dev/block/mmcblk0p2:	1024MB	userdata
/dev/block/mmcblk0p3:	13651MB	sdcard

data. Although we implemented a program to perform multi-pass wipe of the partition, destroying the key alone is adequate as we will be left with a partition full of encrypted data which cannot be decrypted.

V. PERFORMANCE

A. *Experimental Setup*

For our experiments, we use the Google's Nexus S smartphone device with Android version 2.3 (codename *Gingerbread*). The bootloader of the device is unlocked and the device is rooted. The persistent storage on Nexus S smartphones is a 507MB MTD (Memory Technology Device). MTD is neither a block device not a character device, and was designed for flash memory to behave like block devices. In addition to the MTD device, Nexus S has a dedicated MMC (MultiMediaCard, which is also a NAND flash storage technique) device dedicated to *system* and *userdata* partition, which is 512MB and 1024MB respectively. Table II provides the MTD device and MMC device partition layout.

In order to evaluate this setup for performance, we installed two different types of benchmarking tools. We used the SQLite benchmarking application created by RedLicense Labs - RL Benchmark Sqlite. To better understand fine-grained low level file I/O operations under different I/O patterns, we use IOzone [7], which is a popular open source filesystem micro benchmarking tool. It is to be noted that these tools are both a very good case study for 'real-use' as well. RL Benchmark Sqlite behaves as any application that is database-heavy would behave. IOzone uses the direct file I/O intensively just like any application would, if it was reading or writing files to the persistant storage. All other applications which run in memory and use the CPU, graphics, GPS or other device drivers are irrelevant for our storage media tests and the presence of encrypted filesystem will not affect their performance.

IOzone is a filesystem benchmark tool [7]. The benchmark generates and measures a variety of file operations and has been widely used in research work for benchmarking various filesystems on different platforms. The benchmark tests file I/O performance for the generic file operations, such as *Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read, aio_write*.

IOzone has been ported to many platforms and runs under various operating systems. Here in our paper, we use ARM-Linux version (Android compatible) of latest IOzone available and focus on the encryption overhead. The cache effect is eliminated by cold rebooting the device for each run of IOzone and RL Benchmark Sqlite. The device is fully charged and connected to external USB power while in experiments. We collect the data and plot the average results of the 5 runs in the figures in all the following experiments.

B. *Throughput Performance of EncFS*

In this section, we present the IOzone performance results for random read and write operations on userdata partition. The benchmark is run for different file sizes and for each file size, with different record lengths. The maximum file size

Table III
SQLite performance on Google Nexus S

SQL Operation	Time taken by the operation (in seconds)		
	EncFS	**EXT4(No Encfs)**	**Overhead**
100 SELECTs without an index	0.064	0.064	0%
100 SELECTs on a string comparison	0.068	0.059	15%
5000 SELECTs with an index	2.247	2.478	-9%
DB Read(The above 3 operations)	**2.379**	**2.601**	**-8%**
1000 INSERTs	87.25	54.262	61%
25000 INSERTs in a transaction	2.826	2.620	8%
25000 INSERTs into an indexed table in a transaction	2.837	2.628	8%
Creating index	1.198	1.008	19%
1000 UPDATEs without an index	6.607	6.431	3%
25000 UPDATEs with an index	7.018	5.704	23%
INSERTs from a SELECT	3.215	1.932	66%
DELETE without an index	5.300	1.944	173%
DELETE with an index	5.419	1.636	231%
DROP TABLE	4.614	0.921	401%
DB Write(The above 10 operations)	**126.284**	**79.0835**	**60%**
Overall	**128.663**	**81.684**	**58%**

is selected as 4MB due to the observation that 95% of the user data files are smaller than 4MB on a typical Android system.

Fig 3 compares the throughput for four typical file I/O operations, namely *read*, *random read*, *write* and *random write*. The IOzone experiments are run on the original ext4 file system and EncFS with different AES key lengths. Fig 3 shows for *read* operation, EncFS performs the same with original *ext4*. However, for *random read*, *write*, *random write*, EncFS only gives 3%, 5%, 4% of the original throughput respectively. Our analysis shows the encryption/decryption contributes the overhead and is the expected trade-off between security and performance. The buffered read in EncFS makes the *read* operation only incur marginal overhead. However, for *random read*, the need for the data blocks alignment during decryption results in slower throughput. For different key length, the 256-bits key only incurs additional 10% overhead comparing to 128-bits key for better security. In particular, AES-256 runs 12866KB/s,8915KB/s, 9804KB/s at peak for *random read,write and random write* respectively while AES-128 runs 14378KB/s, 9808KB/s, 10922KB/s. The performance loss of a longer key length trading better security properties is only marginal to the performance loss of the encryption scheme. Optimizations can compensate such key-length overhead as illustrated in Section V-D. Based on this observation, AES-256 is recommended and used as default in the following subsection unless otherwise mentioned explicitly.

Similarly, *sdcard* partition gives the identical pattern with slightly different value. Due to the fact that the *sdcard* partition shares the same underlying physical MMC device with *userdata* partition as listed in Table II, our experiment results demonstrates the original vfat filesystem performs 16% faster than ext4 filesystem for *read and random read* operation while ext4 outperforms vfat 80% and 5% for *write and random write* operations respectively. However, comparing different filesystems is out of our focus in this paper. We observed different throughput values and overhead patterns on other devices such as Nexus One, HTC Desire and Dell Streak which use a removable sdcard as separate physical medium to internal NAND device. Both AES-128 and AES-256 throughput on *sdcard* are statistically identical to the ones on *userdata* partition given a 95% confidence interval. Such results show that the scheme of encryption in EncFS(e.g. internal data block size, key length) and its FUSE IO primitives are the bottleneck of the performance regardless of the underlying filesystems. We suggest corresponding optimizations in Section V-D.

In addition to the basic I/O operations, we look at the read operation in detail under different file I/O record size before and after encryption. In particular, we plot the 3D surface view and contour view. In the 3D surface graph, the x-axis is the record size, the y-axis is the throughput in Kilobytes per second, and the z-axis is the file sizes. The contour view presents the distribution of the throughput across different record sizes and file sizes. In a sense, this is a top-view of the 3D surface graph. Figure 4 and 5 show the throughput when IOzone read partial of the file from the beginning. Figure 4 shows the default *ext4* file system in Android 2.3 favors bigger record size and file size for better throughput. The performance peak centers in the top-right corner in the contour view of the 3-D graph. However, after placing EncFS, the performance spike shifts to the diagonal where the record size equals to file size. This is an interesting yet expected result because of the internal alignment of the file blocks in decryption.

To better understand the performance of our encryption filesystem under Android's SQLite IO access pattern, we present the database transactions benchmark in the next subsection, which is more related to the users' experiences.

C. SQLite Performance Benchmarking

In addition to the IOzone micro benchmark results in last subsection, we measure the time for various typical database transactions using the RL Benchmark SQLite Performance Application in the Android market [11]. Table III groups the read and write filesystem operations and lists the results in detail.

We consider that random read and write is a fair representation of database I/O operations in our scenario. This is due to the fact that for SQLite, the database file consists of one or more pages. All reads from and writes to the database file begin at a page boundary and all reads/writes are an integer number of pages in size. Since the exact page is managed by the database engine, file-system only observe random I/O operations.

After incorporating the encryption filesystem, the database-transactions-intensive apps slows down from 81.68 seconds to 128.66 seconds for the list of operations as described in the Table III. The read operations reflected by select database transactions shows the consistent results with IOzone result: the EncFS buffers help the performance. However, any write operations resulting from insert, update, or drop database transactions will incur 3% to 401% overhead. The overall overhead is 58%. This is the trade-off between security and performance.

D. Optimization

To achieve better performance, we optimize the EncFS. The first parameter we tuned is filesystem block size. EncFS support block sizes of 64 to 4096 bytes while 1024 bytes is the default. After we re-create encryption file system and configure 4096 bytes as the block size, we observe performance boost for *random read,write and random write* while the *read* gives statistically equivalent result. The experimental results showed additional 15%, 21% and 19% performance gained for *random read,write and random write* respectively.

Direct-IO gives the throughput for *read, random read,write and random write* as 13197KB/s, 13190KB/s, 11026KB/s, 12689KB/s respectively. Such results gives additional 4%,23%, 28% performance gains for for *random read, write and random write* respectively. However Direct-IO defeats the buffer mechanism in FUSE and causes *read* performance to fall from 472770KB/s to 13197KB/s. As its name indicates, the Direct-IO option makes the filesystem keep pulling and flushing the storage device without buffering, and such continuous writings perform better than buffered writings on encryption filesystem environment. Figure 6 shows the combined performance gains in percentage of the 4k block size and Direct-IO configuration. This experiment on sdcard partition shows the same increased performance pattern.

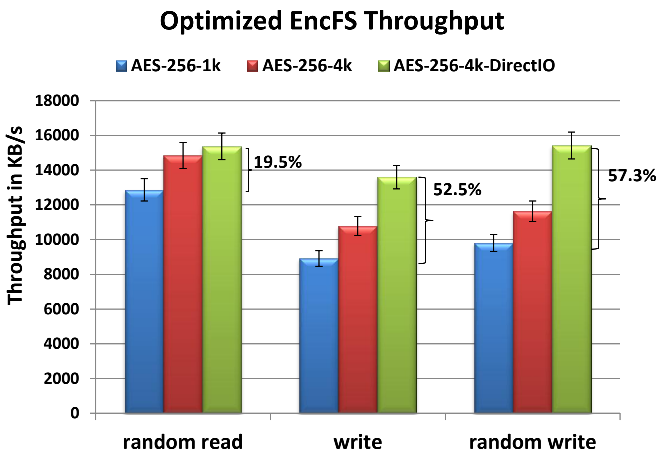

Figure 6. Optimized EncFS Throughput, with 1k/4k Block Size and DirectIO

Table IV
GOOGLE NEXUS ONE DATA PARTITION PERFORMANCE.

Operation	Throughput in KB/s		Overhead
	Yaffs2	EncFS AES-256	
read	318947	311577	2.3%
random read	328834	12752	2479%
write	3455	1559	121.7%
random write	3381	1221	177%

E. Portability and Device Specific Optimization

In addition to the Nexus S device, we also run our EncFS on Nexus One, HTC Desire, and Dell Streak, to demonstrate portability of our approach. Vendors manufacture the devices using diverse hardware modules that come with different feature and qualities. Despite the variance of hardware electrical characteristics and quality, the corresponding kernel driver for such hardware components also play critical role for functionality and performance. For instance, the aforementioned three types of devices all use Qualcomm QSD8250 chipset as CPU. However, only Dell Streak enables the dynamic CPU frequency governing feature in the kernel. Such vendor specific discrepancies cause systematic performance tuning leverage hardware features a challenging task and it can be done only to specific devices. We name a few of our experiences here. Firstly, we run the same benchmark in previous sections on all three types of devices. Our data shows all the three devices using YAFFS2 filesystem give noticeable slower performance than *EXT4* filesystem before encryption. Therefore encryption overhead appears to be less. However, comparing the filesystems or the quality of the physical medium is out of the scope of this paper. We list Nexus One's */data* performance in Table IV.

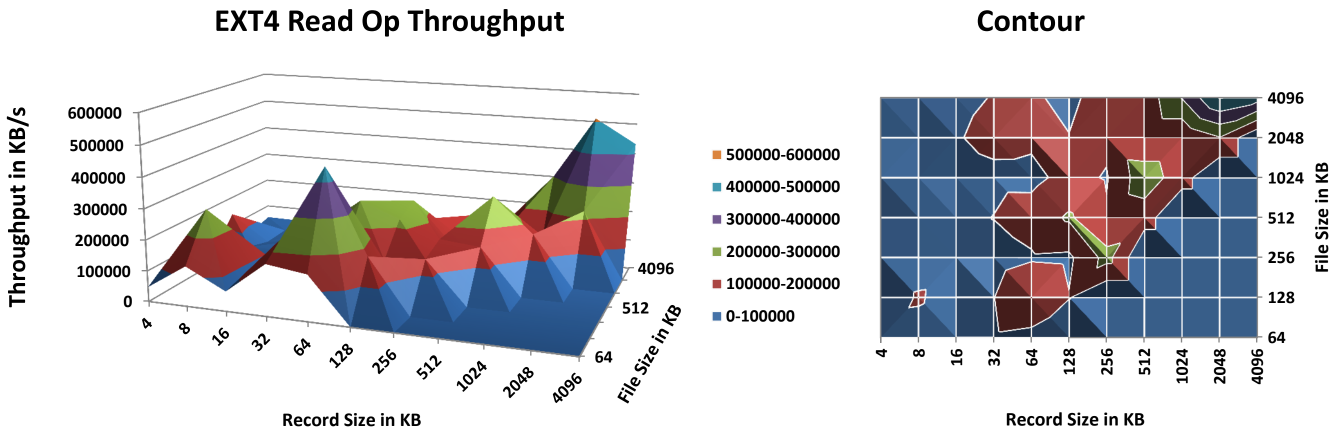

Figure 4. EXT4 Read Operation Throughput

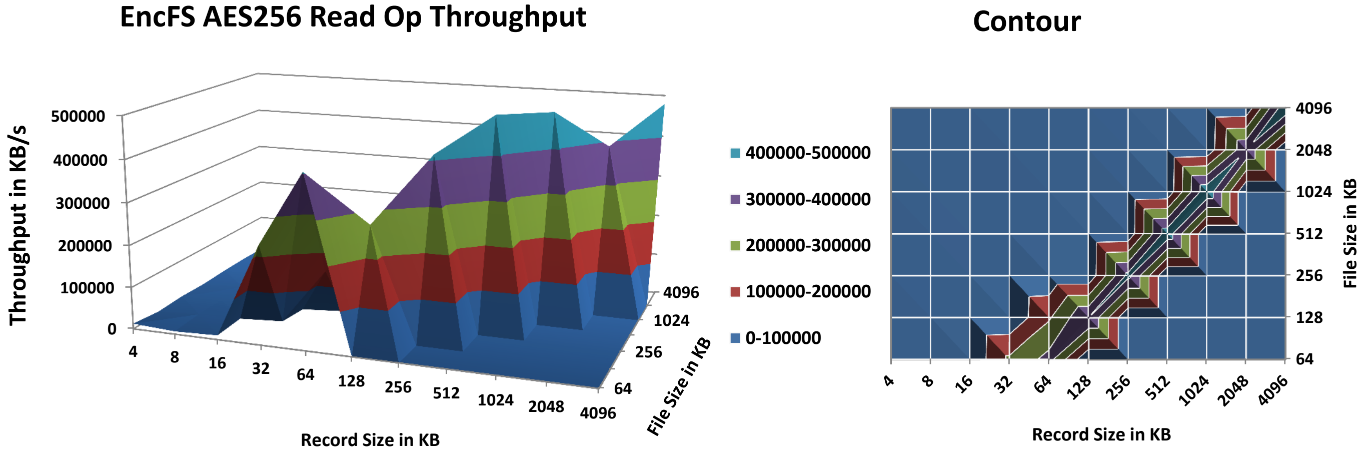

Figure 5. EncFS AES256 Read Operation Throughput

Secondly, for Dell Streak, we over-clock the main CPU frequency from 1GHz to 1.245GHz and gain extra 3% optimization while maintaining the stability of the overall system. In this case, faster encryption computation is the driving source of the additional performance. However, a higher value than 1.245GHz of the CPU main frequency results unstable system behavior.

Thirdly, we tune particular hardware parameters in the kernel driver for additional optimization. Android(Linux) system export a variety of hardware specific configuration through *sysfs*, which allows the user changing the behavior of the hardware. Specifically, we change the default value of */sys/block/mtdblock5/queue/max_sectors_kb* from 8 to 128 on HTC Desire, and gain 10% throughput for read and write operations. Moreover, the default I/O scheduler for this particular type of device is set to BFQ (Budget Fair Queueing). After changing it to CFQ (Completely Fair Queuing) at */sys/block/mtdblock5/queue/scheduler*, an extra 5% performance is obtained. However, for Nexus One and Dell Streak, such entries in *sysfs* are set to be readonly or not available at all. Such tuning are strongly depend on the particular hardware and the specific kernel driver.

In sum, we are able to run regular applications on our EncFS filesystem without any noticeable lag on all above devices, and have deployed a fully encrypted userdata and sdcard partitions on many phones which are in use. Therefore, we conclude that despite a large cost in performance for encryption, this filesystem-level encryption technique is feasible, specially in the case of database intensive applications.

VI. Discussion

In this section, we discuss the limitations of our approach and various discoveries in our experiments.

Filesystem encryption can only protect the data kept on any external or internal storage but not data in the memory or over the network. It can, however, offer protection and prevent data from being accessed externally in the case of lost or captured devices by adversaries. To the best of our knowledge, there is no system with encrypted memory yet that can support all Android devices. Systems with encrypted filesystem are still vulnerable to existing memory-borne attacks such as buffer overflows and requires additional defenses [15]. If the attacker gains system privileges by such attacks, he may not have direct access to the encrypted filesystem but he can access the memory which is in plain text. In addition, encryption filesystem is also vulnerable to cryptographic attacks such as known-plaintext attack or side channel attacks against its cryptographic module.

Based on the fact that the *cache* partition only contains the de-dexed [4] Java code for speeding up the Java runtime and no user specific data in this partition, our approach keeps the *cache* partition as unencrypted, which is consistent to other encryption filesystem approaches[1], [13]. Our focus is only on encrypting the user-specific data. To achieve this, we created encrypted copies of only *userdata* and *sdcard* partitions on Nexus S. However, it is to be noted that our implementation is scalable to any partition that might need encryption. For instance, Dell Streak phones use a separate partition for databases called *firstboot*. We encrypted this partition using the same technique as described for *userdata* partition in this paper.

Surprisingly, we found that Android 3.2's built-in encryption functionality on HTC Flyer tablet is unstable during activation, when the user selects *encrypt tablet*. Further, there is no interface to switch between encrypted or unencrypted states. It can only be activated on *userdata* partition. Once activated, the system reboots into encrypted state and doesn't have the flexibility that a userspace FUSE-based implementation like ours provides. Such evidence indicates the in-kernel approach is not scalable.

Our further investigation reveals, in Honeycomb, zeros were considered encrypted mistakenly. Even if the built-in encryption of Android 3.2 or Android 4.0 is eventually adopted, it does not adhere to the NIST approved cryptographic standards and cannot be deployed for government or military use.

VII. Conclusions

In this paper, we presented an implementation of a portable filesystem encryption engine that uses NIST certified cryptographic algorithms for Android mobile devices. We offer a comparative performance analysis of our encryption engine under different operating conditions and for different loads including file and database (DB) operations. Our experimental results suggest a 20 times overhead for write operations on the internal storage. When increasing the cryptographic key-length from AES-128 to AES-256, we incurred an additional performance loss of 10% to 15%, depending upon the operation performed. Although file operations incurred a 20 times overhead, the database operations had a much more moderate overhead of 58% which accounts for sequential write and update DB operations.

By optimizing the filesystem block-size and I/O mode, we were able to gain 20% to 57% performance. In addition, we then demonstrate that device-specific optimization methods can also provide performance boost. Despite the seemingly large overhead observed for I/O intensive applications, we were successful in running our encryption filesystem on a variety of Android devices and applications without significant user-perceived latency. Therefore, we conclude that our encryption engine is easily portable to any Android device and the overhead due to the encryption scheme is an acceptable trade-off for achieving the confidentiality requirement.

Acknowledgment

We would like to thank Brian Schulte, Sharath Hiremagalore, Quan Jia from the Center for Secure Information Systems at George Mason University for their comments on our early draft. We also thank the anonymous reviewers for their constructive comments. This work was supported in part by the U.S. Army Research Office (ARO) through contract W91INF-II-I-0349 and NSF grant CNS-TC 0915291. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government and sponsoring agencies.

References

[1] "Android honeycomb encryption," http://source.android.com/tech/encryption/android_crypto_implementation.html.

[2] "Android security overview," http://source.android.com/tech/security/index.html. [Online]. Available: http://source.android.com/tech/security/index.html

[3] "Boost c++ library," http://www.boost.org/. [Online]. Available: http://www.boost.org/

[4] "Dalvik format," http://en.wikipedia.org/wiki/Dalvik_(software). [Online]. Available: http://en.wikipedia.org/wiki/Dalvik_(software)

[5] "File system in user space," http://fuse.sourceforge.net (2006). [Online]. Available: http://fuse.sourceforge.net

[6] "Fips pub 1402, security requirements for cryptographic modules." [Online]. Available: http://csrc.nist.gov/publications/fips/fips1402/fips1402.pdf

[7] "iozone," http://www.iozone.org/.

[8] "Librlog," http://www.arg0.net/rlog.

[9] "Nielsen report," http://blog.nielsen.com/nielsenwire/online_mobile/u-s-smartphone-market-whos-the-most-wanted/.

[10] "Openssl fips 1402 security policy, version 1.2."

[11] "Rl benchmark: Sqlite," https://market.android.com/details?id=com.redlicense.benchmark.sqlite.

[12] "Sqlite," http://www.sqlite.org/.

[13] "Whispercore android device encryption," http://whispersys.com/whispercore.html.

[14] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *HotMobile '10: Proceedings of the Eleventh Workshop on Mobile Computing Systems &#38; Applications*. New York, NY, USA: ACM, 2010, pp. 49–54.

[15] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," in *Proceedings of the fourth ACM conference on Wireless network security*, ser. WiSec '11. New York, NY, USA: ACM, 2011, pp. 127–138. [Online]. Available: http://doi.acm.org/10.1145/1998412.1998434

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 296–310.

[17] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Comput. Surv.*, vol. 43, pp. 2:1–2:37, December 2010. [Online]. Available: http://doi.acm.org/10.1145/1824795.1824797

[18] W. Enck, P. Gilbert, B. gon Chun, L. P. C. J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI '10: Proceedings of the 9th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2010, pp. 255–270.

[19] W. Enck and P. McDaniel, "Understanding android's security framework," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 552–561. [Online]. Available: http://siis.cse.psu.edu/android_sec_tutorial.html

[20] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: an auditing file system for theft-prone devices," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966447

[21] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2008, pp. 239–252.

[22] H. Kim, N. Agrawal, and C. Ungureanu, "Examining storage performance on mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, ser. MobiHeld '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:6.

[23] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin, "Secure deletion for nand flash file system," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 1710–1714. [Online]. Available: http://doi.acm.org/10.1145/1363686.1364093

[24] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *RAID '09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 244–264.

[25] B. R. Moyers, J. P. Dunning, R. C. Marchany, and J. G. Tront, "Effects of wi-fi and bluetooth battery exhaustion attacks on mobile devices," in *HICSS '10: Proceedings of the 2010 43rd Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

[26] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring integrity on mobile phone systems," in *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*. New York, NY, USA: ACM, 2008, pp. 155–164.

[27] D. C. Nash, T. L. Martin, D. S. Ha, and M. S. Hsiao, "Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices," in *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 141–145.

[28] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel, "Semantically rich application-centric security in android," in *In ACSAC '09: Annual Computer Security Applications Conference*, 2009.

[29] D. M. Radmilo Racic and H. Chen, "Exploiting mms vulnerabilities to stealthily exhaust mobile phones battery," in *In SecureComm 06*. SECURECOMM, 2006, pp. 1–10.

[30] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 206–213. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774130

[31] N. Tolia and M. Satyanarayanan, "Benchmarks for mobile database access," in *Proceedings of the 1st international workshop on System evaluation for mobile platforms*, ser. MobiEval '07. New York, NY, USA: ACM, 2007, pp. 47–47.

[32] N. Tolia, M. Satyanarayanan, and A. Wolbach, "Improving mobile database access over wide-area networks without degrading consistency," in *Proceedings of the 5th international conference on Mobile systems, applications and services*, ser. MobiSys '07. New York, NY, USA: ACM, 2007, pp. 71–84.

[33] Z. Wang and A. Stavrou, "Exploiting smart-phone usb connectivity for fun and profit," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 357–366. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920314