Exposing Security Risks for Commercial Mobile Devices

Zhaohui Wang, Ryan Johnson, Rahul Murmuria, Angelos Stavrou

Department of Computer Science
George Mason University, Fairfax VA 22030, USA

Abstract. Recent advances in the hardware capabilities of mobile handheld devices have fostered the development of open source operating systems and a wealth of applications for mobile phones and table devices. This new generation of smart devices, including iPhone and Google Android, are powerful enough to accomplish most of the user tasks previously requiring a personal computer. Moreover, mobile devices have access to Personally Identifiable Information (PII) including a full suite of location services, camera, microphone, among others.
In this paper, we discuss the cyber threats that stem from these new smart device capabilities and the online application markets for mobile devices. These threats include malware, data exfiltration, exploitation through USB, and user and data tracking. We will present the ongoing GMU efforts to defend against or mitigate the impact of attacks against mobile devices. Our approaches involve analyzing the source code and binaries of mobile applications, hardening the device by using Kernel-level network and data encryption, and controlling the communication mechanisms for synchronizing the user contents with computers and other phones. We will also explain the enhanced difficulties in dealing with these security issues when the end-goal is to deploy security-enhanced smart phones into military and tactical scenarios. The talk will conclude with a discussion of our current and future research directions.

1 Introduction

The need for smaller, faster, portable devices and the ever increasing use of technology in our everyday life has driven the hardware manufacturers towards hand-held mobile devices that can offer a wide-range of functionality and with affordable cost. Newly developed smart gadget devices are produced by Apple, Google, Samsung, HTC are powerful enough to accomplish most of the tasks that previously required a personal computer. To make matters worse, unlike most desktop or laptop computers, they are almost always connected to the network. 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.

Unfortunately, although powerful and ubiquitous, researchers and practitioners have only recently been looking into the potential threats that stem from

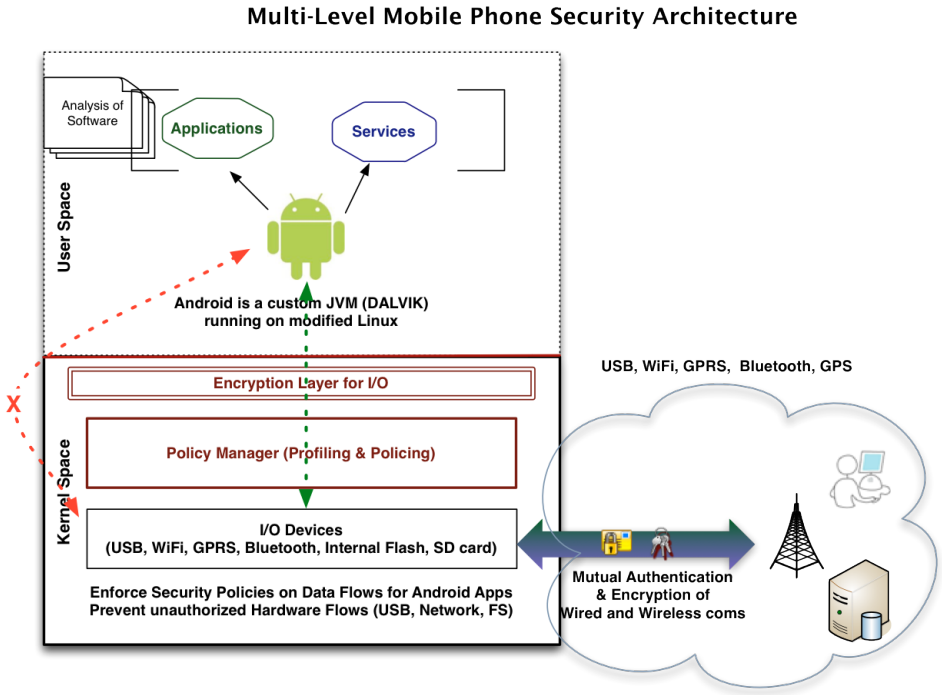

Fig. 1. Overall Security Architecture for Android Devices: the primary design tenet is to prevent Data exfiltration or loss from unauthorized communications and malicious or badly designed mobile applications. A secondary goal was to produce a system that is transparent with small operating footprint in terms of power, CPU, and memory.

device and application attacks on mobile devices. In this paper, we describe the rationale behind some of our efforts [1–3] to secure the hardware and software on Android devices used in adversarial environments. Our efforts are data-centric and is multi-pronged as depicted in Figure 1. One of our primary goals was to provide transparent government grade data-at-rest encryption. An Encrypted File System (EncFS) for Android that employees NIST validated crypto algorithms was employed to meet this need.

On the other hand, we wanted to protect information that enters or leaves the mobile device and to prevent unauthorized data leaks. To achieve that, we employ cryptographic communications to all the allowed paths including the USB communications and Internet connections. Finally, to prevent information leakage from untrusted applications, we developed offline security software testing algorithms for Android applications that enable us to weed-out potentially unwanted program functionality that can be construed as malicious depending on the mission requirements.

All the above solutions, and especially encryption, however, come at a potentially significant performance cost depending on the device we apply them on. In general, on mobile devices resources, including battery and processing power are

severely constrained so it is important to maintain a small operating footprint. Throughout this paper, we show that our proposed solutions as depicted in the overall architecture 1 are offering a reliable and secure platform for deployment of missing critical Android applications even when deployed in hostile or any other high risk environments.

2 Background & Related Work

This section provide some background information on the different research solutions that have been proposed over the last few years and illustrates the difficulties to provide an overarching approach to protecting Android mobile devices against a wide-range of attacks.

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" [4] 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 [5] 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 [6] the authors show how to establish trust and measure the integrity of application on mobile phone systems. TaintDroid [7] 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. Felt et al. [8] performed testing of Android 2.2.1 in order to identify the Android API calls, intents, and content providers which require a permission.

Battery-borne Deny-of-Service: Racic and Kim et al. [9, 10] 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 [11] the authors show that applications can simply overuse the WiFi, Bluetooth or display of the device and eventually cause a denial of service attack. VirusMeter [12] 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 [13].

Mobile Malware and Rootkits: Given the popularity of mobile application and their strong coupling relation with PII (Personal Identifiable Information), the spreading of mobile malware is becoming an alerting threats to military personnel as well as civilians. The evolution of mobile malware created a need to systematically characterize them from various aspects including their installation methods, activation mechanisms as well as the nature of carried malicious payloads given a nearly two years time window [14]. Zhou et al. [15] developed a program to analyze the bytecode of an Android application to create behavioral footprints on Android application and then use heuristics to detect classes of malware.

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. Bickford et al. [17] 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.

Code Injection: Buffer overflows also plague mobile devices. The presentation on hacking Windows Mobile [18] at Xcon 2005 talked shell code development advice as well as sample code. Recent emerging threats show that such exploitations are targeting web browsers and other potentially exploitable software like adobe pdf view application in the mobile OSes. Android platform also exposed multiple vulnerabilities for code injection attacks such as CVE-2011-3874 etc. Bojinov et al. proposed a mechanism of executable ASLR that requires no kernel modifications for defending remote code injection attacks for mobile devices [19].

Static Analysis and Execution: There is a plethora of research on static and dynamic analysis of programs with more notable symbolic execution [20]. Most of the analysis programs focus primarily on determining if an Android application requests the correct set of permissions based on the Android API calls that the applications performs [8]. In addition, static analysis programs usually require access to the source code of the Android application. One of these static analysis programs that executes on source code [21] focuses specifically on certain types of behaviors, vulnerabilities, and limited analysis of the permissions. Vidas et al. [22] created a static analysis tool which detects when an Android application requests permissions that it does note need as well as needed but absent permissions from the AndroidManifest.xml file of and Android application. They also developed a plugin for Eclipse which informs developers when they request unneeded permissions based on the application's functionality. They developed a mapping based on the documentation of the Android API. This documentation for the Android API is incomplete, so their mapping of Android API calls is also incomplete. Blasnig et al. [23] use the Android emulator that comes with the Android SDK to perform dynamic analysis on Android application and use

a tool to simulate user interaction. The tool also performs some static analysis by disassembling the Android application and identifying certain functionality. The random-input generation helps to traverse various paths through the code, although using symbolic execution would inform the random-input generator as exactly what inputs would be needed to reach a particular branch of code that has interesting behavior. Moreover, Burguera et al. [24] also use a sandbox to perform dynamic analysis of Android applications and use a behavior-based approach to classify malware by examining the system calls of that the application makes.

3 Motivation

In this section, we discuss the problems and weaknesses we found while researching commodity mobile systems which leads to our proposed solution in next section.

3.1 Open USB Communication

Traditionally, a smart phone device is connected to the host as a peripheral USB device. Being lack of intelligence and computation power, the device is more prone to be taken over by a compromised computer or abused as malware propagation medium. However, the potential attack surface is much wider: the USB creates a bidirectional communication channel, ideally permitting exploits to traverse both directions. Most USB devices are dumb storage medium or only has limited 8bit computation ability such as keyboard. However, new generation phones are equipped more advanced CPUs with complete operating systems which make them as powerful as a traditional desktop system. These recent hardware advancements enables such USB peripherals to perform attacks that are far beyond their ancestors with no chips in terms of computational and software capabilities. Additionally, unlike desktop computers and servers that do not move their physical location, the mobility nature of the smartphones empowers them to potentially communicate to an even larger number of uninfected devices across a wider range of administrative domains. For example, a smart phone left unattended for a few minutes can be completely subverted and become an point of infection to other devices and computers. Lastly, because USB-borne attacks have not been seen before, there are no defenses in place to prevent them from taking place or even detect them.

Currently, USB connections are inherently trusted by the users. When USB protocol was designed decade ago, the physical proximity of the device and the desktop system attributed to such assumed trust based on the fact that, in most cases, the same user owns both systems. However, Wang et. al demonstrated this trust can be easily abused by a malicious adversary [1]. For instance, an unsuspected user connects the smart phone device to the desktop computer to synchronize the two devices including her contact list, media content, calendar

and to charge its battery. There are several communication setup steps happening in the systems but all of these are performed completely transparently to the user or with minimal user interaction: the simple press of a mouse click upon connecting the USB cable. To make matters worse, the usb host(a desktop computer in most of the case) is completely unaware of the type of the device that is connected to the USB port. The usb peripherals can arbitrarily report itself as any usb device given the crafted usb id. This observation can be exploited by a sophisticated adversary who already gained access of the smartphone to launch attacks against the desktop system. Furthermore, there are no mechanisms to authenticate the validity of the device that attempts to communicate with the host in current USB protocol. The lack of authentication allows the connecting peripheral device to disguise and report itself as any type of USB device it want to be, abusing the ubiquitous nature operating system. While the open-medium problem for bluetooth and WiFi has been address in protocol design phrase so that the communication are protected, the USB communication implying a closed-medium do still has the open-medium problem given that the two parties of the communication can not authenticate each other. Our goal is protecting the devices as well as the host from such attacks by applying access control mechanisms on the USB protocol. We refer the USB host as the host system or host side while the USB device as gadget or just device side in the following sections of this paper.

3.2 Lack of Protection for Data at rest

The recent surge in popularity of smart handheld devices, including smartphones and tablets, has given rise to new challenges in protection of Personal Identifiable Information (PII). 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 such as GPS, bluetooth and WiFi, the data collected and generated are extraordinarily sensitive to user's privacy. Indeed, modern mobile devices store PII for applications that span from daily emails to SMS, and from social sharings to location history increasing the concerns of the end user's privacy. Smartphones are therefore data-centric, 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. Therefore, there is a clear need and demand for PII data to be protected in the case of loss, theft, or capture of the hardware.

While the application sandboxing protects application-specific data from being accessed by other applications on the phone, sensitive data may be intentionally exfiltrated by malicious code via one of the communication channels such as USB, WiFi, Bluetooth, NFC, cellular network etc. It also can be leaked

accidentally due to improper placement, resale or disposal of the device and its storage media (e.g. removable sdcard). 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. There is no government standard to regulate and guide the use of smart devices yet. Given the cheap price and rapid evolution 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 through its entire lifetime, we must have robust techniques such as encryption to store and delete data while keeping confidentiality.

We assume that an adversary is already in control of the device or the bare storage media in our threat model, . The memory-borne attacks and defences are not discussed in this paper and addressed by related researches in Section 2 and discussed later. 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. Our encryption filesystem protects the static data on storage in complimentary to dynamic information flow leaking [7]. Having established a threat model and listed our assumptions, we detail the steps to build encryption filesystem on Android in the following sections.

3.3 Missing Fine-Grain Application Auditing and Regulation

Permission Model on Android platform created debated situation in both industry and academic community. Is such permission model really capable of regulating the applications on mobile operating system and protecting average user's data?There are a couple of studies showing that such permission model, as scatter throughout the whole Android API, failed the aforementioned design goal. In particular, mobile malwares have made their way to bypass such permission model by using other existing applications' capabilities to delegate the malicious behavior. In another work, good app with legit permissions can behave bad. Furthermore, malicious app can also utilizing reflection [25] to evade the permission checking system. Moreover, current permission system is a bipolar system: the user can either grant *all* or deny *all* permissions asked by an application. Such inflexible approach impeded the advanced user to fine-grain auditing and regulating the behavior of the application. We believe that a proper static and dynamic analysis infrastructure will assist both smartphone users and application developers to understand the applications' footprint on filesystems, network and other subsystems. The analysis results can lead to malware discovery and better application design.

4 Proposed Solutions

In previous section, we exposed the missing component in commodity mobile systems. We address those problems by proposing our solutions to tackle them in this section.

4.1 USBSec: Authentication for USB Communication

We outline the design principles we follow and the detailed design of two types of USBSec, a passcode approach and a public key based approach.

Design Principles Our principle is using easiest engineering, modify minimum set of USB protocol, to achieve reasonable security enhancement including identity authentication, connection authorization. Our design philosophies for both USBSec I and USBSec II are outlined as follows:

- *The authentication is device driver agnostic.* There are a variety of different USB host controllers and USB peripheral controllers in the consumer's market. The design does not depend on any specific device or device driver to accomplish our goal. The authentication logic happens at USB protocol level so that any host controller driver or peripheral controller should have such USB authentication capability when a modified OS kernel with USBSec loads up.
- *No USB hardware modification.* Although the modification is at USB protocol level, the hardware signalling and interrupts remain intact. Only operating system software level changes make the deployment process of USBSec time and cost efficient.
- *Our design is backward compatible.* All existing USB hardware can be used as normal if the peripheral are not listed as USBSec required. The host selectively activates USBSec by configuration listed device vendorID and productsID, when it initiates the USB connection. This is critical to those non-programmable devices which implements USB protocol in the hardware, i.e. USB keyboard and mouse. Our modification to USB enumeration process is compatible with all standard USB devices. In another word, standard USB handshake proceeds if the peripheral device is not listed as authentication required.
- *The authentication is per device.* In the case of a composite USB gadget device, multiple interfaces are available for communication with the host at the same time. The per-device authentication design guarantees that no interface can performance potential malicious action until the authentication of the device is finished.

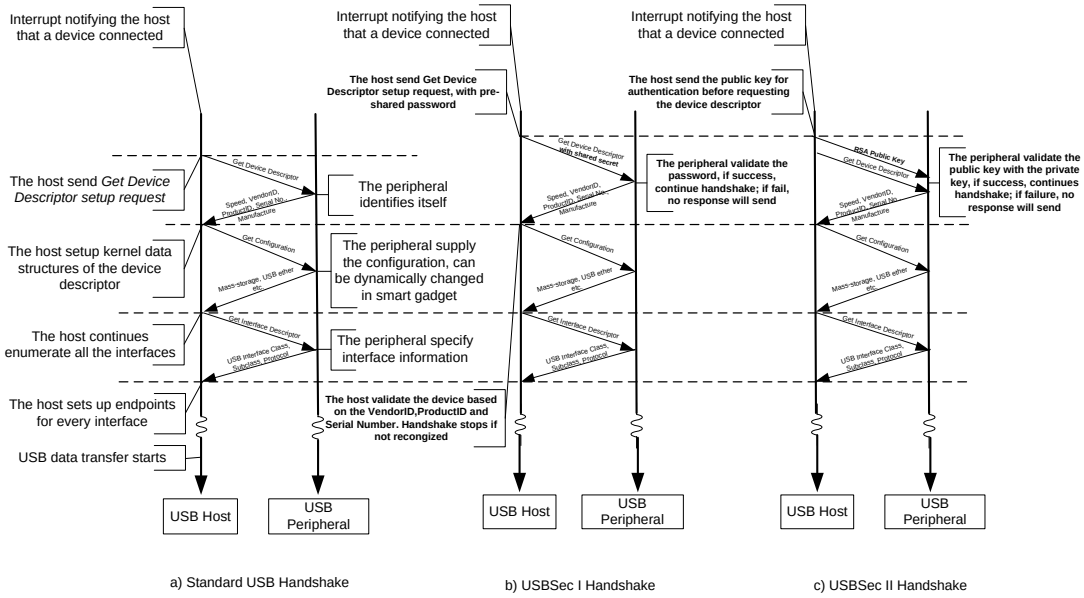

Fig. 2. USB Handshake Diagrams.

Implementation We have implemented a fully working prototype of USBSec I and USBSec II. Our evaluation platform consists of a Dell PowerEdge 1950 server equipped with two Quad-Core Intel Xeon E5430 processors, 16GB RAM as the host; a HTC NexusOne phone and a Motorola Droid phone as the devices. We evaluated USBSec on both devices to show that our design and implementation are not tied to specific hardware controller. The host has the Intel 631xESB/632xESB/3100 chipset as the USB host controller. The devices have the msm_72k OTG controller integrated in QSD8250 SoC with a 1 GHz CPU and ISP1301 USB OTG controller integrated in the OMAP 3430 SoC with a 600 MHz CPU respectively. Both devices run a 2.6.32 based android kernel with respect of different SoC support. The host is running Ubuntu 10.04 with a 2.6.32 kernel. Although we have user-space programs to set the necessary configuration for USBSec, all USBSec processing logic are implemented tightly with the existing USB stack in the Linux kernel. For instance, in USBSec I, the shared passcode is configured via */proc* filesystem by user-space utility, both on the host and the device. Similarly in USBSec II, the keys are generated by *openssl* suite version 0.9.8k program and stored in file system. We wrote the user-space daemon program to load the keys when system boots up and pass it to kernel data structures. Unlike USBSec I, the passcode can be changed at system runtime, the keys in USBSec II only loads at system bootstrapping and only have corresponding kernel memory footprint at runtime. In another word, reconfiguration of the keys requires the system reboot. Specifically, our user-space daemon program will decode the PEM format of the public and private keys to DER(binary) format using *Base64* decoding algorithms and pass it to pre-allocated data struc-

tures in kernel. The kernel is responsible to do the Diffie-Hellman key exchange using asymmetric crypto primitives to establish the session keys. However, in mainstream kernel, there is no RSA functionality support yet. We merged the existing RSA kernel implementation [26] with our additional modification to accept the DER binary format keys as input in the gadget kernel to achieve in kernel RSA cipher support. The key size is selected as 1024 bits to tradeoff moderate cryptographic security with performance.

We performed experiments in order to quantify the performances and compare it with traditional USB connection scheme. We measure the connection setup time of USBSec I and USBSec II against standard USB. The start of connection time is define as the time that the host receives the interrupt from USB receptacle notifying the kernel that a device is being connected. The end of connection time is defined as the time that the host finishes learning the device descriptor and starts requesting the configuration information. Both of them are indicated by a kernel log entry. This time interval includes all our authentication and validation extensions to USB stack. Figure 3 shows our experiments result of USBSec extensions to USB protocol on the two different devices. The analysis result of the data can be highlighted as follows:

- Our USBSec extension only incurs very small amount overhead in connection time and do not affect users' experience. Both of the devices complete the handshake within 2 seconds.
- Handshake takes different amount of time on difference devices, due to different USB controller model. The results reveal that NexusOne takes less time accomplishing the USB handshake than Droid, even in standard USB protocol. USB peripheral controller hardware and the device driver cause the difference.
- Major CPU frequency also plays a key role in the Diffie-Hellman key exchange to establish asymmetric keys. The worst case scenarios reveal that the 1 GHz NexusOne performs faster than the 600 MHz Droid for the same amount of iterations, 256 times in our case.

USB Connection Time(ms)	Standard	USBSec I		USBSec II	
		Best Case	Worst Case	Best Case	Worst Case
NexusOne	254.6ms	343.1	346.7ms	471.1ms	578.9ms
Moto Droid	322.0ms	1373.8ms	1378.2	1744.0ms	1908.6ms

Fig. 3. USBSec Connection Time

Discussion Linux kernel uses a single bit to disable/enable a USB device on the USB bus [27], providing a basic authorization machanism. In depth, the kernel will set the device descriptor to "n/a (unauthorized)" and disable it by removing the device configuration information. However, the this scheme has fundamental

flaws. First of all, all wired USB devices are authorized by default. In addition , such authorization happens only after the device being connected to the host and the host already enumerated all the interfaces in the devices on the USB bus. It require human interactive operation to explicitly deauthorize the device afterwards. The malicious program running on the device has more than enough time to compromise the host during this gap. Furthermore, this scheme can only authorize the device on the host. From the device's point of view, there is no mechanism to authenticate the identity of the host. For example, a smartphone containing sensitive information can not defend itself from being connected to an unidentified host. Moreover, experiments show that when the user disconnect a deauthorized USB device, the kernel panics at usb_disable_endpoint function and the system become unusable. Further kernel code investigation reveals that even when the same device being connected to the host again, it will be authorized by default.

SELinux applies security policies labeling to files, and AppArmor applies the policies to pathnames. None of them take considerations for devices inside the kernel.

USB 3.0 is planned to allow for device-initiated communications towards the host, which will make the things more complicated for implementing the authentication scheme in the USB stack. However, we believe with moderate engineering effort, the device-initiated communication can also be authenticated by our approach.

Implementation is crucial to the security strength in any crypto system. It has been conclusively shown that textbook RSA is insecure [28, 29]. Secure RSA requires that padding scheme must be used before encryption and signing. USBSec II's in kernel RSA implements padding functionality to the basic RSA operations to encrypt(), decrypt(), sign() and verify() methods.

For passive devices that only have USB hardware implementation but no CPU(e.g. storage device, keyboard), full mutual authentication can not be accomplished due to the limitation of computation capability at the device. Nevertheless, we can authenticate the device and logical driver information by the serial number after the first packets exchange, and prompt the user at the host to allow or deny the connection. Such allowance or rejection can be temporary or permanent. In most of the cases, it's difficult to spoof the serial number information in such passive devices. End user's knowledge and approval help secure the connectivity.

Limitation Like any password based approach, USBSec I is facing brute force attacks. The adversary can exhaust the password space and defeat USBSec I if gained control of the kernel of either side of the USB communication. The second limitation is USBSec I authentication use serial number information specifically tied to the hardware as the gadget side identity. Any hardware or new inventory change will need corresponding updates to the authorized devices whitelist on the host side.

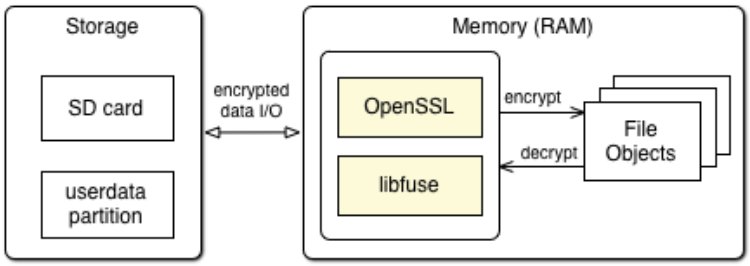

Fig. 4. Abstraction of Encryption Filesystem on Android

Bear in mind that we are protecting unauthorized access, USBSec is defeated if the host or the device is already being compromised and spoofs the identity.Because at that time, the trusted chain is broken and authentication is useless.

4.2 EncFS for Android

EncFS is selected as the basis for our encryption filesystem.

EncFS is composed of three major components : kernel FUSE library support, user space *libfuse*, and EncFS binaries. In addition, to make an encryption file-system work on Android, a modified bootstrapping and user login was also integrated into the Android operating system.

EncFS uses standard OpenSSL cryptographic libraries in userspace as cryptographic primitives. 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 [30, 31] can be summarized 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 multiple hardware vendors, only minimal engineering efforts if any are needed to make EncFS work on other variations of Android-based smart devices.
- EncFS leverages OpenSSL FIPS suite as the crypto service engine. The OpenSSL libraries, namely libcrypto and libssl, implement cryptographic algorithms that are validated with government FIPS 140-2 Level:1 standard [32].
- In addition, our approach supports different underlying file-systems transparently, including *yaffs2*, *ext4* and *vfat*.

To build EncFS for Android, we created a package with the components described below. It is required the phone has root access at installation time in order to accommodate the kernel with FUSE support, system binaries and

java framework patches for integrated login. Once installed, EncFS does not need processes or applications to run as root, in order to encrypt the data. The applications work transparently without knowing any change underlying changes.

Kernel FUSE support: FUSE module provides a bridge to the actual kernel interfaces in general. However, the Android Linux kernel does not support FUSE file-systems in early versions. Such minimal kernel configuration reduces fileystem and memory footprints on mobile devices and also eliminate redundant functionalities that are not required by Android. For instance, most Android devices, including the Nexus S which we use, do not come with the FUSE modules enabled in the kernel in off-the-shelf state. We obtain the kernel source code from Google's Android Open Source Project (AOSP) and enabled the kernel FUSE modules necessary for *libfuse* to run. We then flash our device with this customized kernel.

Libfuse: As the required supportive library for all FUSE-based file-systems, *libfuse* is not officially included or supported in the Android system. Moreover, the Bionic C library in Android is a trimmed version of C libarary and 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 executables for the ARM architecture, we created the binaries that would enable us config and manage the EncFS filesystem. In addition to *libfuse*, EncFS also depends on the boost library which is a widely adopted C++ library[33], *librlog* for logging[34] 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.

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 as standard block cipher by US government, our experiments focus on AES only.

Depending on whether we built them as static or shared libraries, we push the binaries onto the system binaries locations on the phone. Figure 4.2 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 the user-specific data. In our system configuration for EncFS, this */data* partition keeps only a skeleton of the minimal folders to make system bootstrapping. We store the encrypted data in a separate directory and mount it over */data* mountpoint when the user supplies the password.

In addition, we modified the Android Launcher application 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

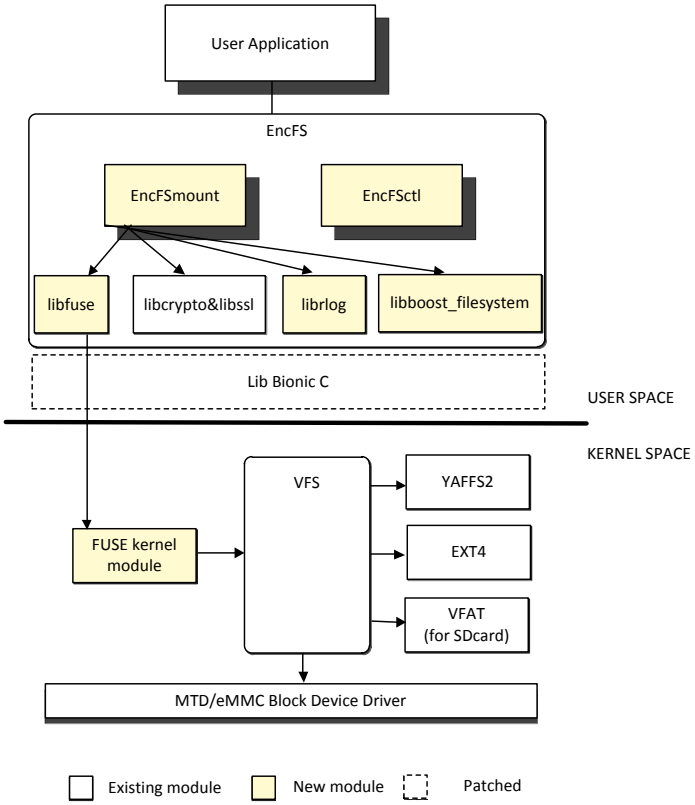

Fig. 5. The operational layout of the Encrypted File System (EncFS).

partition on */data* mountpoint. If such mount is performed successfully, the Launcher will call a dedicated native program installed by us to *soft* reinitialize the Android Dalvik environment and the user is presented with his encrypted userdata partition decrypted and loaded into the memory transparently.

To avoid brute-force attacks against the password, the user has a 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 encrypted data using secure deleting mechanisms. Although we implemented a program to perform multi-pass secure 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.

Implementation and Performance Please refer to our full paper [3] for performance details and optimizations.

4.3 Application Analysis

Static analysis serves as a useful method to examine the possible behavior that a program can exhibit. However, static analysis is constrained to certain functionality due to its inherent limitations of not actually executing the code [35]. Static analysis is susceptible to false positives, false negatives, and obfuscation [36, 37]. The precision of the analysis increases when the analysis program better understands the semantics of the code and is able to observe the state of the program. When using dynamic analysis, test inputs need to be randomly generated, come from a pre-generated set, or be input by an active entity. Dynamic analysis may or may not get complete coverage of the code, but all the instructions executed will be reachable and the program's true behavior can be observed.

To get as close as possible to complete coverage of the code, a method must exist to affect the control flow of the program. As each conditional statement is encountered, either the values of the variables would need to be changed at runtime to obtain the desired outcome, determined a priori by symbolic analysis, or be forced by controlling the jump to a particular branch independent of the outcome of the boolean condition being evaluated. This type of execution approach [38, 39] stresses the program by entering as many branches as possible to make the program exhibit different types of behavior.

The impetus behind this approach is to maximize the coverage of code, as opposed to examining the behavior of the program exhibited by a more limited number of execution traces. This is important because malware can contain very specific conditions that must be met in order to for it to display malicious behavior [40]. In certain instances, the behavior is triggered by certain events such as specific times, dates, hostnames, local IP addresses, the presence of a file, and other factors. In addition, a program may restrain its malicious functionality when it determines that it is being debugged, running in an emulator, or some other type of controlled execution enivornment [41].

We have developed a set of tools, that runs on a computer and performs concrete execution of an Android application while abstracting certain details from the execution of the application. This abstraction allows the program to automate the analysis of as many paths as possible through the application without requiring any user input. Due to the abstraction, automation is achieved, but the precision of the analysis is reduced. The abstraction is necessary due to not running the application on an Android-enabled phone and the absence of the Android API in disassembled applications. The program only requires an Android Package (apk) file which is the compressed format used to encapsulate an entire Android application into a single file.

Our program uses apktool [42] to disassemble an apk file to obtain Dalvik bytecode. Execution of the Dalvik bytecode is possible because we created a Java implementation for each Dalvik bytecode instruction. After using apktool, the disassembled application will be missing the code for the Android API since this code is resident on all Android-enabled phones. Therefore, calls to the Android API are not actually executed unless they are specifically handled by our program which only occurs for a very small set of simple API calls. These were

manually coded into our program after examining the specification and exhibited functionality for each handled API call. In addition, we also leverage the Java API since the program runs inside the JVM by creating a wrapper around certain Java API calls.

Our program enables the values of primitive data types and objects used as parameters to Android API calls to be examined and extracted. The values of the parameters depend on the specific execution path taken through the program. The values of the primitive data types within an object that are used as a parameter to an API call can also be examined. This enables a more through understanding of the program's behavior, and allows hard-coded values in the application to be operated on and extracted. The precision of the analysis is limited due to the lack of user interaction, user input, and Android API which will prevent certain values and objects from being available.

For example, there is an Android application with the package name of com.antivirus.kav which uses a simple technique to obfuscate the actual domain that it connects to. The method called LinkAntivirus returns an domain that has been transformed using three calls to the replace method of the java.lang.String class. Figure 1 shows the smali [43] for the Dalvik bytecode which upon conclusion will ultimately return a String containing http://routingsms.com/z.php. Further examination of the smali code reveals that the application will connect to this domain using the openConnection method of the java.net.URL class. The application also appends the phone number, the device ID, and the subsriber ID of the phone to this domain which occurs in the GetRequest method of the SmsReceiver class. Using static analysis on all the files of the disassembled application would not be able to detect the obfuscated domain, as well as the phone-specific information it appends to the domain.

The program allows all Dalvik instructions, method calls, and API calls to be hooked for analysis and monitoring. Currently, the program monitors and records very specific behavior (e.g., commands issued, execution of binaries, use of Java reflection, loading of libraries, network events, files accessed, exhaustive list of methods called, control flow, and the use of dynamic class-loading), although this can be further expanded to anything of interest. The use of Java reflection is recorded because it can help to obfuscate the actual method being called unless the parameters are examined to see exactly which method of what class is being called. Reflection also obviates the use of visibility modifiers in Java [25].

Limitations Our approach can be computationally expensive depending on the structure and size of the program being analyzed. Each if conditional statement that occurs outside of a loop exponentially raises the number of iterations that must be executed to cover all possible paths within an application. Loops that are nested to a high degree significantly affect the performance of the program due to the large number of iterations through the code, especially when many if conditional statements occur within each loop. An attacker could purposefully plant various computationally expensive activities throughout the program to purposefully slow the analysis of the application. There are approaches to make

a trade-off between analysis time and analysis precision. There are the options to limit the number of iterations through a loop, prevent nesting beyond a certain number of loops, limit recursion, or use a timer that sets a maximum time that can elapse to indicate that a path should end.

The program is unable to obtain input entered by the user. We use an abstract representation for the objects that are returned from Android API calls which are not specifically handled by our program. These objects will have the same corresponding type but they will have no state. The state can be built as operations from the application set the value of the instance variables of an object. The program also does not have support for multithreading. There are limitations to our approach due to the entropy in environment variables, user input, and non-deterministic routines.

The program will enter all branches even if they are unreachable. The unreachable branches, conditional statements that will always be false, can result from programming logic errors. Upon execution of the program, the branch will never be entered when it is executed without forcing the outcome of a boolean conditional associated with a if conditional statement. As the program encounters a if conditional statement, it will execute both branches of an if conditional statement without consideration if the boolean condition can never be true. This could result in a false positive when looking for certain behavior if it occurs in an unreachable branch.

5 Conclusion

We presented a framework for exploring all available paths in an Android application. We verified our approach by testing a large number of Android applications with our program to exhibit its functionality and viability. The framework allows complete automation of the process, so that no user input is required. We plan to overcome the limitation of the absence of the Android API and phone hardware by incorporating an Android-enabled phone to handle some of the processing. The program can also be used for program validation by determining what behavior can be exhibited by the application. The program also has various user set parameters that enable a trade-off between performance and the precision of the analysis. The program can serve as a an extensible basis to serve other useful purposes such as an interactive debugger, symbolic execution, dynamic analysis, and any other approach that requires analysis of an Android application to be performed on a computer.

References

1. Wang, Z., Stavrou, A.: Exploiting smart-phone usb connectivity for fun and profit. In: In ACSAC '10: Annual Computer Security Applications Conference. (2010)
2. Wang, Z., Johnson, R., Stavrou, A.: Attestation & authentication for usb communications. In: IEEE International Conference on Mobile Data Management (IEEE MDM 2012). (2012)

3. Wang, Z., Murmuria, R., Stavrou, A.: Implementing & optimizing an encryption file system on android. In: In SERE '12: 6th International Conference on Software Security and Reliability (SERE 2012). (2012)
4. Enck, W., McDaniel, P.: 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) 552–561
5. Ongtang, M., Mclaughlin, S., Enck, W., Mcdaniel, P.: Semantically rich application-centric security in android. In: In ACSAC '09: Annual Computer Security Applications Conference. (2009)
6. Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B.M., Jaeger, T.: 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) 155–164
7. Enck, W., Gilbert, P., gon Chun, B., Jung, L.P.C.J., McDaniel, P., Sheth, A.N.: 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) 255–270
8. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In Chen, Y., Danezis, G., Shmatikov, V., eds.: ACM Conference on Computer and Communications Security, ACM (2011) 627–638
9. Radmilo Racic, D.M., Chen, H.: Exploiting mms vulnerabilities to stealthily exhaust mobile phones battery. In: In SecureComm 06, SECURECOMM (2006) 1–10
10. Kim, H., Smith, J., Shin, K.G.: 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) 239–252
11. Moyers, B.R., Dunning, J.P., Marchany, R.C., Tront, J.G.: 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) 1–9
12. Liu, L., Yan, G., Zhang, X., Chen, S.: 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) 244–264
13. Nash, D.C., Martin, T.L., Ha, D.S., Hsiao, M.S.: 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) 141–145
14. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2012) 95–109
15. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: In Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS 12. (February 2012)
16. David, F.M., Chan, E.M., Carlyle, J.C., Campbell, R.H.: 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) 296–310

17. Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., Iftode, L.: Rootkits on smart phones: attacks, implications and opportunities. In: HotMobile '10: Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications, New York, NY, USA, ACM (2010) 49–54
18. Phrack: Hacking windows ce. http://www.phrack.org/issues.html?issue=63&id=6
19. Bojinov, H., Boneh, D., Cannings, T.R., Malchev, I.: Address space randomization for mobile devices. In: Fourth ACM Conference on Wireless Network Security (WISEC 2011). (2011) 127–138 http://www.odysci.com/article/1010113016076341.
20. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (July 1976) 385–394
21. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proceedings of the 20th USENIX Security Symposium. (August 2011)
22. Vidas, T., Christin, N., Cranor, L.: Curbing Android permission creep. In: Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011), Oakland, CA (May 2011)
23. Bl andsing, T., Batyuk, L., Schmidt, A.D., Camtepe, S., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on. (oct. 2010) 55 –62
24. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. SPSM '11, New York, NY, USA, ACM (2011) 15–26
25. Richmond, M., Noble, J.: Reflections on remote reflection. In: Proceedings of the 24th Australasian conference on Computer science. ACSC '01, Washington, DC, USA, IEEE Computer Society (2001) 163–170
26. Linux: Rsa kernel patch. http://lwn.net/Articles/228892/
27. Community, L.O.S.: Linux usb authorization. http://lxr.linux.no/linux+v2.6.32.24/Documentation/usb/authorization.txt
28. Boneh, D., Cryptosystem, T.R., Rivest, I.R., Shamir, A., Adleman, L., Rst, W.: Twenty years of attacks on the rsa cryptosystem. Notices of the AMS **46** (1999) 203–213
29. Bellare, M., Rogaway, P.: Optimal asymmetric encryption how to encrypt with rsa. In: Advances in Cryptology Eurocrypt, Springer-Verlag (1995) 92–111
30. Team, G.A.: Android honeycomb encryption. http://source.android.com/tech/encryption/android_crypto_implementation.html
31. Whispercore: Whispercore android device encryption. http://whispersys.com/whispercore.html
32. Project, O.: Openssl fips 140-2 security policy
33. Boost: Boost c++ library. http://www.boost.org/
34. Librlog: Librlog. http://www.arg0.net/rlog
35. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: NDSS, The Internet Society (2008)
36. Chess, B., McGraw, G.: Static analysis for security. IEEE Security and Privacy **2**(6) (November 2004) 76–79
37. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. (dec. 2007) 421 –430

38. Wilhelm, J., Chiueh, T.c.: A forced sampled execution approach to kernel rootkit identification. In: Proceedings of the 10th international conference on Recent advances in intrusion detection. RAID'07, Berlin, Heidelberg, Springer-Verlag (2007) 219–235
39. Lu, S., Zhou, P., Liu, W., Zhou, Y., Torrellas, J.: Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In: In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO. (2006)
40. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically Identifying Trigger-based Behavior in Malware. In: Botnet Analysis. Springer Publications (2007)
41. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2007) 231–245
42. android apktool: A tool for reengineering android apk files. `http://code.google.com/p/android-apktool/`
43. Smali: An assemble/disassembler for android's dex format. `http://code.google.com/p/smali/`