The Pennsylvania State University
The Graduate School

INTERRUPT AND IPC DRIVEN KERNEL FRAMEWORK FOR PREVENTION AGAINST SMARTPHONE MALWARE

A Thesis in
Computer Science and Engineering
by
Ashwin Chaugule

© 2009 Ashwin Chaugule

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Ashwin Chaugule was reviewed and approved* by the following:

Sencun Zhu
Assistant Professor Computer Science and Engineering & IST
Thesis Advisor

Bhuvan Urgaonkar
Assistant Professor Computer Science and Engineering

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Chair of Graduate Program of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Smartphones have several network interfaces like WiFi, Bluetooth and GSM. Since today's telephony infrastructure supports 3G and various other protocols of data transfer like WAP, GPRS and EVDO, it is possible to bring the desktop internet experience on the handheld device. With this comes the same level of security risk that we see on desktop machines. There is extensive research [1],[2],[3] on securing such desktops. But there is only a recent effort in securing handheld devices. The computing power of such devices restricts the portability of security solutions from the desktop over to embedded systems. Due to these restrictions, there is a need to optimize the security solutions without compromising on the effectiveness.

The key idea behind the solutions presented here is to detect a real users intent to trigger an event such as sending an SMS or making a phone call. Malware which attempts to perform these events tries to do so without the knowledge of the user and hence the events triggered are purely generated in software. The way to differentiate these events is to monitor the hardware interrupts generated by the keypad or touchscreen of the device, since it is the only way a real user can begin an event such as sending a message or making a call. Our framework resides entirely in the kernel and adopts a specification based prevention approach. The specification is defined by signatures of application behavior using their $inter-process$ *communication patterns. Using the hardware interrupt as a necessary event for any function of the device, followed by a signature defined communication pattern, we are able to prevent two of the most common attacks (messaging and covert audio channel attacks) found on mobile devices. The framework is lightweight and has*

almost negligible overheads (20 μs) during the normal functionality of the running system.

Table of Contents

List of Figures vii

List of Tables viii

Acknowledgments ix

Chapter 1
Introduction 1
1.1 Smartphones . . . 1
1.2 Thesis Contribution . . . 2
1.3 Structure . . . 3

Chapter 2
Security Model 5
2.1 Threat Model . . . 5
2.2 Attack Model . . . 6
2.3 Trusted Computing Base . . . 7

Chapter 3
Background 9
3.1 Linux . . . 9
3.2 Subsystems . . . 10
3.2.0.1 Serial Subsystem . . . 10
3.2.0.2 Touchscreen Interrupt . . . 11
3.2.0.3 Input Device . . . 12
3.2.0.4 Interprocess Communication . . . 12
3.2.0.5 Cryptography Subsystem . . . 13
3.2.0.6 Timer Subsytem . . . 13
3.2.0.7 Process Control Block . . . 14
3.2.1 Audio Subsystem . . . 14
3.3 Openmoko . . . 14
3.4 Qtopia . . . 15

Chapter 4
Related Work 17

Chapter 5
Design 22
5.1 Model Overview 22
5.2 Qtopia Model 24
5.3 Formalization 26
5.3.1 Signatures 27
5.4 Reference Monitor Design 29
5.4.1 Reference Monitor Interfaces 29
5.4.2 Authorization Module 32
5.5 Reference Monitor State Machine 34
5.5.1 States 34
5.5.2 Reverse Edges 36
5.6 Reference Monitor Assessment 36
5.6.1 Complete Mediation 37
5.6.2 Tamperproof 38
5.6.3 Verifiability 39

Chapter 6
Results 40
6.1 Application Text Segment Sizes 41
6.2 Application Load Time 41
6.3 Training Run Time 42
6.4 Input Event Overheads 42
6.5 IPC Overheads 43

Chapter 7
Conclusion 45

Appendix A
Code Fragments 48
A.1 TEXT Segment Hashing 48

Appendix B
Files Affected 51
B.1 Git Diff 51

Bibliography 53

List of Figures

5.1 Qtopia Stack Diagram 25
5.2 Reference Monitor 30
5.3 State Machine 35

List of Tables

3.1 Sample AT Command Set 11

5.1 Messaging Truth Table 27
5.2 Audio Call Truth Table 28

6.1 Applications TEXT Segment Sizes 41
6.2 Application Load Overheads 41
6.3 Training Run Overheads 42
6.4 Input Event Overheads 42
6.5 IPC Overheads 43

Acknowledgments

I would like to thank Dr. Sencun Zhu for giving me the constant motivation and support to pursue this research. I am very grateful to him for giving me the freedom to choose my topic and explore my interests. My sincere thanks to Dr. Bhuvan Urgaonkar for providing his inputs on my work. I would also like to extend my thanks to my colleagues to bear with me during our technical discussions.

Chapter 1

Introduction

1.1 Smartphones

Since the early 90's we have seen several devices which were labelled as smartphones of that era. The Palmtops, Symbians, Nokias all had their go at making sleek and slender devices that would bring your daily productivity applications such as calendar, alarms, todo lists, planners etc on the palm of your hand. While the technology to make things faster existed, it was too expensive to bring it to the consumer. Therefore these devices of the early age were limited in what they could achieve. However, once the embedded systems industry picked up pace, we saw several high power ARM processor variants make their way into such handheld devices. These low cost, low power consuming devices marked a new beginning for the handheld industry. Developers could port their desktop applications on these devices with relative ease. More processing power meant more eye-candy applications and low power consumption meant longer durability. This was exactly what the consumer market needed.

Today we see a lot more advancement on the handheld hardware. Devices such as the Apple iphone, Google gphone, Openmoko [4]etc. have advanced chipsets that can handle GSM/3G telephony, wireless networking and much more. Some even have support for accelerated graphics. With the beginning of GSM capability on these handhelds, the market for smartphones became very lucrative. Bigger companies such as Microsoft, Apple, Windriver, Google started to build frameworks that enabled users of their devices to develop applications themselves. Some companies such as Apple and Google even have a shared online market place for individual developers

to sell their applications online to other users of their phones. Meanwhile, the Linux market also showed its strengths. The open source model that had already existed and matured for so long made it an easy transition for Linux to be ported on these devices. The ease with which one could customize their devices with everything from the kernel to the running applications made Linux a strong competitor for this market.

While the market for handheld devices was growing rapidly, it is only now, that we are seeing a stronger focus on the reliability issues of these devices. The security issues of desktops were already well known and some of them were solved. But since these devices enabled the cross porting of desktop applications onto the phones, we see similar vulnerability issues being carried over. Security frameworks such as SELinux etc. demonstrated what they could do for hardening desktops. But they could not be easily ported over to embedded devices mostly because of performance issues [5], [6]. This meant we needed other means of securing the mobile application space.

1.2 Thesis Contribution

This thesis focuses on preventing the most common attack vectors [7], [8] found in today's smartphones. Past research and surveys [9], [10] shows the alarming number of malware spread through the Messaging interfaces found on the devices. The framework explained in the following chapters aims to prevent the spread of malware by focusing on the root of the problem. The problem in question being unmonitored access to the GSM engine on the device. So the framework detects if a real user intends to send the message by monitoring the touchscreen or keypad interrupts and the ensuing the IPC between critical userspace components. The framework monitors for these specific events and then allows or disallows the outgoing SMS.

Similarly, another attack vector that has been recently brought into the limelight, is the covert channel audio attack. Here, the malware covertly turns on the microphone on the device and records conversations into a file without the knowledge of the device owner. Malware may also interfere with an ongoing call by injecting noise into the headset speaker, or play a music file so the other party hears it instead of the real users voice. A similar attack is possible using the on-board camera. The recorded video or audio file is then transferred back to the attacker through the messaging interfaces. The kernel framework described in this thesis, uses the same

concept as the messaging attack prevention. By monitoring the keypad or touchscreen interrupts and the ensuing IPC, the reference monitor allows or denies access to the audio subsystem.

This kernel reference monitor is a lightweight module that monitors only the specific kernel API that are involved in the messaging and audio related functions. Unlike other security solutions, there is no userspace component that communicates with the kernel to gather data. In this way, the entire userspace stack can be untrusted. The requirements for this design to work are described in the Security Model chapter.

By using malware prevention instead of detection, the kernel framework is able to have only minimal overheads. The main reason for this is that for mobile devices, the attack vectors have a common pattern for attacking. Unlike desktops the applications running on the mobile device are smaller and simpler. The limited resources on the device limit what the malware can do. Every phone stack has some critical components that perform functions on behalf of the other applications. Therefore it is easier to focus on these critical components and monitor their behavior for correctness. Also, if the malware subverts these components and attempts to access the kernel API directly, it can be easily detected by the reference monitor within the kernel. Alternately, using intrusion detection techniques and anomaly detection and misuse detection principles, leads to an extensive database of signatures. These signatures consist of white lists and black lists for application behavior. Also, the monitoring overheads are significantly higher [11] with such approaches which make it an unattractive solution for mobile devices mainly due to the machine learning techniques used by these approaches. By using specification based techniques, we are able to manually define signatures for normal behavior and use the Reference Monitor to enforce this behavior. Although anomaly detection techniques may be able to detect newer forms of attack, we think for mobile devices, the attack vectors are limited. Thus it is very easy to manually add the specifications and we get the benefits of very low run-time overheads.

The results section shows the minimal overheads involved in the monitoring framework.

1.3 Structure

Chapter 2 begins with a description of the Security Model which includes the Threat Model and Trusted Computing Base, Chapter 3 lays out the background information required to understand the technical details of the implementations, Chapter 4 lists the related work, Chapter 5 describes

the main design of the implementation with a formal analyses and Chapter 6 shows the benefits of our design in the form of results and instrumentation. We end with Chapter 7 as a conclusion summarizing the motivations for our design and some possible future research ideas.

Chapter 2

Security Model

Since most mobile phone vendors have started online market places to enable any user to develop his applications and sell them to other users of the device, this may profit the company to market its devices, but it opens up a huge scope for buggy and malicious applications to enter these devices. As an example consider a scenario where a user downloads an application that has a buffer overflow vulnerability. However, this application may still be downloaded by millions of users since the vulnerability doesnt affect the usability of the device and is therefore hidden from the user. Now another malicious developer who is aware of this vulnerability creates another viral application that exploits the other applications buffer overflow. In this way, the malware can spread itself like an internet worm. As another example, the malware may even exploit the affected devices functionality, by depleting its power resources and sending messages [12], [13].

This motivates us to consider the security model relevant to the mobile phone. The components of the model are described below.

2.1 Threat Model

Most of the vulnerabilities in applications are due to programming errors or bugs that are left undetected even after testing procedures. These bugs are most commonly buffer overflows due to lack of out-of-bound error checks. Applications that are coded by inexperienced programmers tend to have such loopholes which are exploited by malicious software. The main components of userspace that are vulnerable constitute the following:

- **Network Interfaces:** Most smartphones comprise of WiFi, Bluetooth the GSM interfaces. The malware enters the device through these interfaces. However, for this thesis we do not consider the ingress path of malware. This can be tackled by looking at the security measures designed around the interaction of the device with the external network infrastructure like access points, securing the network medium on the air and GSM telephony setups of service providers.
- **Phone Stack on the device:** This is the multithreaded application running in user space and is responsible for implementation of the phone functionality. The threads that are at threat are particularly the GSM thread which implements the AT command set and the audio thread that interfaces with the audio subsystem of the kernel.
- **Kernel exported interfaces:** The critical components of the phone stack are responsible for interfacing with the kernel for registering, receiving and sending events and data. For the keypad device the kernel exports a node via the input subsystem (eg. /dev/input/keypad), the GSM engine as a serial node (eg. /dev/ttySx) and the audio port as device nodes (eg. /dev/asound/). These interfaces interact with the hardware through their respective device drivers.

2.2 Attack Model

Given that the userspace applications are part of the threat model, here we consider how the attacker can exploit the interfaces to launch his attacks.

- The attacker can interface directly with the exported serial port of the GSM engine and implement his own messaging framework, thus bypassing all the phone stack components. This can allow him to send messages and interfere with GSM calls.
- Similarly, the attacker can interface with the network components of the system. eg. He could use the BT stack, WiFi network interface to send files to other devices. This is a common technique to spread malware and infect surrounding phones. It can also be used to infect the device with new malware.
- Any other peripherals on the device, like camera, audio ports maybe exported to userspace by their respective device drivers. These peripherals are exported through device nodes

and can be configured through system calls (IOCTL's etc). The telephony stack on the device registers with these nodes and provides the respective service to other applications. But, the kernel doesn't naturally restrict opening the device nodes multiple times. Hence, an attacker may open them by himself and use them to alter the audio configurations by subverting the userpsace stack.

2.3 Trusted Computing Base

Here we describe all the relevant modules that are part of the TCB and the requirements for correct operation.

The operating system running on the device mainly forms the trusted computing base. The security of the operating system as far as rootkits are concerned is assumed in this case. By preventing access to the kernel memory, the installation of rootkits at runtime can be prevented. The alteration of kernel control flow can also be avoided by preventing such modification to the kernel memory [14]. Rootkit installation through malicious device drivers [15] , [16] can be prevented by denying the privileges of the root user, thus allowing only the phone manufacturer to install new modules by flashing the device. There are several other schemes to prevent kernel rootkits. But this is not the focus of this research.

The other components of the TCB are as follows:

- The device driver for the keypad is trusted to perform its operation correctly. The keypad driver correctly initializes the keypad hardware and receives the hardware interrupts. The logic to detect the precise scan code when an interrupt is triggered and the interface to the input device subsystem to pass on the scan code is trusted to function properly. The input device subsystem itself should report scan codes correctly to the user space application. It is up to the user application to interpret this event appropriately.

- The GSM device driver is trusted to implement the bus protocol correctly. This bus interface maybe a serial SPI, I2C or any other proprietary bus, but it needs to ensure the hardware on the other end of the bus is initialized properly. The ensuing transactions over the bus should be interpreted properly since these comprise of data in the form of requests or responses. Although the commands to the hardware maybe implemented by the GSM

thread from the user space which is part of the threat model as described above, the driver must ensure correctness of operations for low level communication. This means the driver is agnostic to the semantics of the AT command set and just deals with the low level bits that are transferred over the bus as per the bus protocol.

- The audio driver should correctly perform its functions of delivering audio and recording audio from the hardware ports. The audio subsystem is also responsible for multiplexing between various input sources such as microphone, headset microphone and the bluetooth handset microphone. The other output sinks include, handset speakers, headset speakers and bluetooth headset speaker. The configuration for these sources and sinks comes from userspace and thus maybe misused by malware. But as described below, the kernel reference monitor can verify the operation.

- The kernel memory interfaces are not exported to userspace via /dev/mem and /dev/kmem. This is a necessary requirement to prevent userspace apps from writing into kernel memory [17]. These interface can be easily disabled during kernel configuration time. They are usually enabled for only debugging purposes.

- The operating system is trusted to perform all its operations on memory, I/O, scheduling, system resource management etc. Particularly the OS is trusted to maintain a correct Process Control Block with the required in-memory representation of the running processes. (eg. Page tables)

- There is a trusted user/administrator of the device who is required for setting up the code page hash of the trusted applications and also for entering the signatures in the reference monitor. This requirement is elaborated in chapter 5.

- The bootloader should correctly load the kernel which contains the Reference Monitor implementation. However, we do not address secure booting in this research.

Chapter 3

Background

This chapter explains all the relevant components involved in the smartphone used for this thesis.

3.1 Linux

Linux is an open source operating system that is developed under a GPL license. The GPL license allows anyone to freely download, extend and distribute the code. Under this principle of development Linux has gained a widespread popularity in the embedded systems, servers and desktop market. The mainline kernel on www.kernel.org is owned by Linus Torvalds who is the founder of the Linux OS. This tree contains all the stable releases of code. The development life cycle for Linus's tree last for 3 weeks per version revision. This means, Linus accepts patches which are deemed stable and worthwhile by him and select few respected developers. These select few developers are core subsystem maintainers who have their own trees and their own schedules for release. eg. Andrew Morton owns the Memory management tree, Dave Miller owns the Networking tree etc. Each of these developers accepts patches to their trees wait till they get stable testing reports from people who use their trees and then submit these patches to Linus's tree.

Any user can download this code and choose to use it and improve it. If the user wants to submit his code upstream, then there is a procedure to submit patches of code to the appropriate lists. These mailing lists are extremely active high traffic lists. The submitted code get almost instant reviews and suggestions and will undergo extensive testing on individual machines. After

a few months, it may get pulled into one of the subsystem trees as mentioned earlier. This suggests a very robust testing procedure. Writing code for the kernel is not a very easy task and is therefore likely to be buggy if not reviewed properly. The Linux open source model helps massively control the kind of bugs that creep in and this is mainly the reason why it has gained widespread popularity as compared to the Windows kernels.

The Linux kernel source code is designed into architecture dependent and independent sections [17]. The architecture dependent parts program the hardware required for initial bootup and hardware bringup, while the architecture independent parts implement the functions of an operating system such as memory management algorithms, I/O management etc. The whole kernel source is supported by the GNU compiler toolchain which is also open source. The compiler contains support for GNU C extensions which are widely used throughout the kernel source code to facilitate extremely efficient and compact code.

Although the choice of operating system here is Linux, the same design principles apply to the other operating systems ported for mobile devices.

3.2 Subsystems

The operating systems functions are implemented as different subsystems. eg. Memory management subsystem, I/O resource management subsystems etc.. The ones that are relevant are described below.

3.2.0.1 Serial Subsystem

The serial protocol is widely used by many hardware devices that are usually low bandwidth (as compared to Ethernet) low error rate tolerant devices. Such devices work on the UART bus. For embedded systems like mobile phones, there are several such UART's. One of those is used for exporting the console (command line shell) to a remote computer like a desktop. This assists in debugging the system as well as testing the applications and kernel code. The other UART ports are used to connect the GSM engine, hardware debuggers etc.. There are faster variants of the UART bus such as SPI bus's that are used for network devices, but their underlying protocol of communication remains the same.

The GSM engine is a separate hardware module that is connected over the serial bus to the

main CPU. The implementation of this serial communication is shared with that of the console serial port. The two are differentiated based on the minor number of the node that is created in the /dev directory by the kernel. This minor number helps the kernel identify which port is communicating at that point in time. The GSM engine is controlled by a set of AT commands. Some important AT commands are described in the table 3.2.0.1.

Command	Description
AT+CFUN	Init hardware
AT+COPS	Get network status
AT+CMGF	SMS mode
AT+CMGS	Submit SMS
AT+ATE0	Echo AT commands

Table 3.1. Sample AT Command Set

3.2.0.2 Touchscreen Interrupt

The interrupt subsystems support an asynchronous way of communicating to the core processor to indicate the occurrence of an event. Since interrupt signals only last a very short period of time (eg. a few nanosecs, or one clock cycle) they cannot convey much data by themselves. Hence they they work as a means to signal the CPU that data is ready. The process of scheduling to obtain that data from the hardware then follows in a separate transaction. However, the main advantage of interrupts is that, their asynchronous nature allows the main CPU to either sleep in power saving mode or attend to other tasks while waiting for data to arrive [18].

For mobile phones one of the most frequently used interrupt based devices is the touchscreen or keypad. The alternative to interrupt driven devices is polling based which keeps the CPU busy polling the hardware, checking for new data. Although this may simplify the hardware circuitry, there are many drawbacks from a power saving and performance perspective. When a key is pressed, the hardware constructs a bitmask indicating which key was pressed. This scan code is conveyed to the main CPU through an interrupt that is raised by the keypad/touchscreen controller.

The device driver for the touchscreen implements the interrupt service routine(ISR). This ISR is invoked by the kernel after the responsible hardware wakes the CPU. The ISR is called in what is called as an interrupt context which is given the highest priority of scheduling in an operating system. This enables fast service time to hardware events. The ISR itself needs to

execute quickly if nested interrupts are not supported. This ensures minimum interrupt latency. The touchscreen ISR is mainly responsible for obtaining the scan code from the hardware. This transfer of data from the hardware memory into CPU memory is the primary job of most ISR's. From then on, the appropriate processing of the data which is in CPU memory is scheduled at a later time. This division of interrupt processing is called top and bottom half processing. The bottom half takes care of the time critical function of the interrupt such as transfer of data from device memory to RAM, and the top half then processes that data into the appropriate subsystem of the kernel.

3.2.0.3 Input Device

The Input Device framework is a consolidated framework in Linux for all input devices like keyboards, mouse, touchscreens, joysticks etc. This unification helps to have a common programming model for userspace applications that register to receive or initiate events. Events are described by a common structure that contains the event scan code, device information and other device specific information. This way, an application that has registered to listen to these device events can choose to filter information as it needs and simultaneously support multiple kinds of input devices.

The touchscreen driver is implemented such that it hooks with the input framework. The ISR of the touchscreen acquires the scan code and delivers it to the input framework for post interrupt processing. In this part of the code, the scanned code is converted into an input event by filling up the necessary fields. The input event is then enqueued into a buffer and the userspace application is woken up through an asynchronous I/O notifier signal. The woken up application then performs a read system call on the device node when it is scheduled after the signal. The event is then transfered from the kernel buffer to the userspace application memory. The scan code to key bindings is performed by a key map that is part of the user application. This way the same scan code can perform different application specific functions depending upon the context.

3.2.0.4 Interprocess Communication

Interprocess communication is implemented in several ways in an operating system. Processes can communicate through sockets that belong to the AF_UNIX family, share pages in physical memory, use pipes to transfer data and so on. The common thing in all these mechanisms is

that the data transfer from one process address space to another occurs only inside the process context in the kernel. The process context is one where the kernel is performing some functions on behalf of the process.

For the socket based communication, the usual socket initialization procedure is followed, wherein the server sets up a socket and the client registers or binds to that socket. The only difference from network based socket communication is that there are no IP addresses or ports involved in the socket descriptor. The socket is a named socket node that looks like a file. The open, close, read, write system calls work according the POSIX standards and require a socket file descriptor. The system call implementation in the kernel recognizes the socket as a AF_UNIX family socket from the socket descriptor and appropriately invokes the send_msg/read_msg functions for IPC communication. The sockets are associated with peers which signify the other end of the socket communication. This helps the kernel identify the destination of the data during a send call and also the source of the data during a receive call. Each socket has an associated queue for the data in the process address space that owns the socket. During the send/receive calls the data is enqueued during the send call and the peer is notified of data ready in queue. This invokes the dequeue operation on the socket after a read call.

3.2.0.5 Cryptography Subsystem

The crypto subsystem in the kernel is a new framework that implements the famous hashing and encryption/decryption algorithms. The idea of consolidating it in a framework has the benefit that the algorithm is agnostic to the other subsystems using it. Therefore, network stacks IPsec and disk based crypto can share the same code and not interfere with each others functionality.

The crypto system supports encryption of chunks of bytes from different regions of memory through scatter gather lists. These lists are usually used by DMA engines. The data is picked up from various memory addresses and represented as a chunk of data to the crypto algorithm. This avoids the need to copy data from different memory and redundant buffers.

3.2.0.6 Timer Subsytem

The kernel supports multishot and single shot timers. These are mechanisms to run parts of code at a deterministic point in the future. The timer resets itself and runs again for the next iteration for multishot timers. Timers are used for polling events. They are especially useful if

data needs to be processed at specific intervals like in voice and video codecs. It can also be used to check the status of flags or variables periodically. Timers run in interrupt context so the caveats for programming ISRs apply to them as well.

3.2.0.7 Process Control Block

The process control block is a kernel data structure that describes a process. It contains a lot of metadata about the running processes. The PCB helps to quickly and conveniently organize process specific information in the kernel for scheduling and identifying processes. The PCB is called the task_struct in the Linux operating system. Some important members of task_struct include pid, virtual address ranges of the code, stack and data segments, memory descriptor for the processes page tables, the file descriptors in use by the process and various instrumenting information that indicates whether this task is I/O bound or CPU bound.

3.2.1 Audio Subsystem

The audio subsystem in Linux is called the ALSA (Advanced Linux Sound Architecture) subsystem. The deprecated OSS (Open Sound System) has been replaced by ALSA and provides far more superior features and support for modern audio chipsets. The ALSA framework implements the support for various codecs for wav, pcm, stereo etc. The recording and output functionality and their respective configuration is exported by ALSA to the userspace app's through device nodes (/dev/asound/..). The main advantage of ALSA is that it can interface with the bluetooth kernel stack and provide audio profile support. The real controls to change parameters such as gain, volume, mute etc. are implemented in audio chipset device driver. ALSA just provides an abstraction so that other kernel components and user space applications can interface with the chipset.

3.3 Openmoko

Following the open source model, the FIC company in Taiwan decided to build a completely open source mobile phone based on the open source software. They also decided to open source the hardware reference manuals to encourage the community to contribute and expand the functionality of the phone. The phone is shipped with the tools and debuggers required to open it up and

modify absolutely any part of the phone. The smartphone[19] contains a touchscreens interface, ARM9 based S3C24xx series processor. The core boots off a flash chip and has expandable flash slots. The other features include GSM, GPRS, BT 2.0, AGPS. The kernel core support for the Samsung family of ARM cores is already well maintained online. There are some more kernel patches required to the get other hardware running on the platform. These patches are available on the openmoko wiki pages. Following the linux kernel cycle of development, the openmoko teams have several developers who own their GIT trees with the lastest patches merged. The active community develops and optimizes the main upstream kernels to perform well on the openmoko platform.

The userspace telephony stack also has various options. The first one from the developers at FIC was completely based on GTK+. This distribution had most of the phone functionality running, but lacked some important features like SMS messaging and generally had bugs. The later revisions decided to shift to a different programming model. Instead of GTK+ they used Python bindings with C programming. There were several distributions other than this one. A long list of distributions and their respective howto instructions are described on the wiki page.

3.4 Qtopia

Trolltech is a Finnish company now owned by Nokia, which produced an alternative programming environment for GUI's and desktop applications. This stack is completely programmed in the C++ language. Qtopia is the base for running the KDE desktop environment just as GTK+ is the base for running the Gnome desktop environment. Trolltech then released a new version of their stack for embedded systems called Qt-Embedded [20]. This was primarily not meant to be under an open source license. This made it harder to port Qt-embedded to various embedded devices. However, Trolltech developers seemed to port their stacks to some mobile phones. Around early 2004 Qt-embedded was ported to some PDA's and Motorola phones. Nokia bought over Trolltech and then released the stacks under the GPL licenses. This move allowed a lot more developers to contribute and expand the portability of the software. The Object Oriented model of the Qtopia stack allowed for easy plugin and application development, but the enormity of the code base made it a much harder deal to modify the core components of the stack.

However, with the support of the Qtopia core developers and the open source community,

several applications were coded that made a complete phone stack. This stack supported by the far the most comprehensive features in a phone stack. The standard features included, address book, complete telephony support, messaging, camera support, networking etc. They also extended the programming model to incorporate some security policy support as well.

Chapter 4

Related Work

Securing data and applications on mobile devices is only recently gaining the attention it needs. However, most of the research work focused on optimizing desktop solutions for embedded devices. Very few of these propositions started the design from the ground up with an exclusive focus on mobile devices. Since the mobile devices have a very different usage model and are restricted by their relative lower power resources, there are significant changes in the approaches towards intrusion detection and prevention for cellphones.

Bose, Hu et al. [21] proposed a solution to logically order the events caused by applications on the device. Using a Support Vector Machine they use machine learning theory to detect the pattern of these events and compare them to a whitelist of behavior signatures. This kind of anomaly detection approach works well for them since they generalize the pattern signature in order to keep them lightweight. However, their scheme requires a complex framework in userspace to detect and monitor these learning patterns. Hence they depend on remote analyses of behavior to reduce the overhead of computation on the device.

Guo, Wang et al. [22] layout the most common attack scenarios on mobile devices by categorizing their attack vectors. They show how the blocking probability in the GSM cell can be increased from 0.01% to 1.2%. This shows the attacks caused by infected phones on the backend telephony infrastructure leading to various denial of service cases. Their solutions include shutting down the PC interfacing components of the smartphone during voice calls and messaging to reduce the chances of malware entering the device during a PC sync phase, or enabling the operating system API to turn on the LCD of the device whenever there is an outgoing call or

message to alert the user of suspicious activity and using secure TPM TCG measures to access the SIM card.

Cheng, Wong, Yang et al. [23] developed a collaborative virus detection and alert system for smartphones. They rely on collecting data from neighboring devices and performing joint analyses to detect an infected phone. There is also a provision for a proxy server in cases where collaborative analyses is not possible. This scheme requires an additional component in userspace that constantly monitors for other devices and also collects information from the device it is running on. Also there is a need for constant network connectivity and the requirement that nearby devices need to run their client stub for collaboration.

Traynor, McDaniel et al. [24] [25]demonstrate how malware on phones can exploit the data channels of the GSM network and compromise the infrastructure for whole cities. Specially crafted SMS's packets can flood the channel originating from the internet or users devices and jam the communication channel for voice thus increasing the blocking probabilities.

Schmidt, Bye, Clausen et al. [26] describe an architecture for collaborative anomaly detection on the Android. Their work is a first of its kind for the phones developed with the open source model. However, their scheme revolves around userspace components constantly gathering application and kernel data and depending upon remote servers or peer devices for analyses. The data collected involves systems events such as filesystem changes, signals, process creation, application access events and so on. This data is used to construct behavior patterns and then later for pattern matching at the server or shared amongst other participating devices for analyses.

Forrest and Longstaff [27] showed a way to profile application behavior based on the systems calls it makes. By having an observation window they characterized the normal behavior of an application. They observed that a longer window had a better precision of profiling an application and thereby reduces the false positive rates during comparison at run time. However, this approach has a major drawback. Mimicry attacks are easily able to subvert the detection system. Also, they ignore the arguments passed to the system calls which could be used to improve the identification of a process.

Vigna, Mulliner et al [28] showed a way of labeling processes and data to prevent cross service attacks. By tagging resources used by processes upon network activity they monitor the flow of data in between processes. Their scheme is effective in achieving their goal, but not without overheads. Labeling resources requires several rules, transition of labeling requires monitoring

overheads and false positives can easily occur when legitimate processes initiate network activity.

Kinder, Jha et al. [29] came up with a scheme to normalize malware that obfuscates application binaries. By normalizing the applications before they are executed, the code obfuscations, packing and junk injection vectors of attack are removed. However, this approach can be subverted by malware easily by using novel techniques to exploit vulnerabilities in applications. Also, they are unable to normalize complex executables which contain opaque predicates and branch functions. This makes their approach limited to only a certain Instruction Set Architecture and unsuitable for mobile devices.

Courtney, Stevens et al. [30] categorized and quantified the effectiveness of mobile phone virus response mechanisms. They show the propagation statistics of MMS based malware and categorize the vectors according to point of reception, point of infection and point of dissemination. Their work shows the significance of prevention of malware that originates through MMS's on mobile handhelds. They consider running anti virus algorithms on MMS gateways, the effectiveness of user awareness about malware and monitoring for anomalous activity in the network using the GSM infrastructure elements such as HLR, VLR, BS etc.

Racic, Ma, Chen [13] studied the MMS based malware that depletes energy resources on the phone. Here they stress on exploiting the insecure interaction between cellular data networks and internet (PDP context retention and paging channel). By making use of the active PDP context which remains valid for very long even after the user has finished his transaction, they show that the energy resources can be depleted 22 times faster. Their solutions include making changes to the GSM protocols by reducing the PDP context activation periods, information hiding at the gateways, and making use of the GGSN to discriminate between malicious and normal packets.

Xie, Zhu et al. [31] ported SELinux on the smartphone to prevent the SMS/MMS related attacks. Due to excessive complexities and overheads involved in setting and enforcing the policies, they tried using CAPTCHA's before sending a message. Although CAPTCHA's is an effective way to differentiate between humans and machines, it is an unnecessary inconvenience to the user of the device.

Desmet, Joosen et al. [32] describe a way to securely run third party applications on mobile phones without using the conventional sandboxing techniques. Their design uses secure execution techniques like run-time monitoring, static verification and proof carrying code. They describe a policy package consisting of a policy language to define the security intentions of the application.

The run time monitors insert hooks into the applications and enforce correctness according to the policies. The only problem with this approach is with the policies. The number of policies can increase greatly for critical applications which are responsible for core functionality of the device. Even for simple applications, the policies could be imprecise and thus lead to weaker security.

Trolltech [33] has an article describing Qtopia's SXE(Safe Execution Environment) architecture which works along with the Linux Intrusion Detection (LIDS) kernel patches. This architecture provides a sandboxing framework in userspace for untrusted qtopia plugins. It relies on defining a policy request for the services that the applications require. At run-time the SXE framework monitors for illegal resource accesses. During the installation of that application, the user is given options to accept of deny the policy requests for resources. This solution requires a knowledgeable user and thus can be very un-userfriendly.

Venugopal [34] came up with a faster way to lookup signatures using hashes. He focused on the overhead of detection which signature matches the current behavior in the system. Instead of scanning through the database, a whole signature at a time, he proposed scanning using only a part of the signature. His results show a good improvement in memory and time complexity. However, these anomaly detection techniques still have the downside of false positives and lack of detection of zero day attacks.

Schmidt, Peters, Lamour et al. [35] propose an anomaly detection framework for the Symbian OS based smartphones. Their design learns to extract features from applications run-time behavior which describe the state of the device at that point in time. These features are then sent to a remote server for complex intrusion detection. Such approaches depend upon the intervals for capturing the snapshot of the system. Malware could learn about these monitoring patterns and stealthily subvert the IDS.

Venugopal, Hu, Roman [36] describe a virus detection system for the Symbian platform which monitors the DLL functions used by applications. By using Bayesian decision theory and past virus samples, they try to check the behavior of applications to find matches with malicious activity. Although they claim a 0% false negative rate, they are only able to detect 95% of the viruses.

Kim, Smith et al. [12] aim to detect malware whose main goal is to deplete battery resources. By collecting samples intermittently they form a power consumption signature and later analyze

it to detect for malicious activity.

Divya, Sawani et al. [37] use a stripped down SELinux policy infrastructure based on the PRIMA model to define policies for applications running on the smartphone. Using these stripped down policies for all applications during installation, they are able to prove to remote verifiers that the monitored system is secure enough to run critical third party applications like banking software. However, they still rely on defining policies manually, where improper settings could easily compromise the system.

Xu, Zhang et al [38] described a novel attack which stealthily captures video using the on-board camera found on smartphones. Their algorithm covertly records video according to the phone usage and uses a compression algorithm to store the video on disk. This file can later be transfered to the attacker. These attacks are very realistic and go easily unnoticed to the user of the device. However, they do not propose any solutions. We believe our kernel framework is able to address this issue successfully. Although the Neo1973 device used for this research does not have an on-board camera, the same concept is shown using the microphone device to covertly record audio conversations.

Chapter 5

Design

5.1 Model Overview

For the demonstration of this framework, we consider two kinds of attacks.

- **Messaging Attacks:** Here, the malware attempts to generate a stream of messages by itself with an intention to deplete the battery charge on the phone, spread the malware to other devices and affect the monthly bill of the user.
- **Audio Attacks:** Here, the malware interferes with the audio subsystem of the phone and can act as a covert channel to record the voice conversation or deny the voice conversation by routing the audio source from the MIC to a file. The recording can begin during a call, or even when the user is not using the phone. The latter scenario can be used by the attacker to stealthily capture the ambient sounds of two people conversing.

The common thing between these attacks is that the malware will stealthily try to perform its actions without the knowledge of the user, by directly accessing the hardware. eg. malware will directly try to control the GSM engine and send messages. In order to prevent this, we use specification based prevention methods. We define a specification which comprises of a legitimate series of events in the system which are necessary for the messaging or audio function. This specification is defined as a signature of hardware interrupt generated scan codes and a pattern of IPC between applications. Since the applications involved during the function of sending a message or controlling audio are limited, the specification is very simple, making the signature

very concise. The Reference Monitor enforces the behavior between applications according to the signature.

The implementation of this framework was done in the Linux kernel version 2.6.24 on the Openmoko Neo1973 smartphone with the Qtopia telephony stack. Telephony stacks are usually designed in two different ways.

- The first kind implements a central server that mediates all functionality from other components of the phone stack. For eg. The server will communicate with the Operating System on behalf of the Messaging thread, or even the Networking thread. Qtopia is an example of such an architecture. The advantage of this design is that most of the complexity is hidden away in the central server. This helps to keep the overall architecture simple to develop and expand. Functionality is implemented in the form of plugins which run as threads in userspace. These threads interact with the central server through IPC mechanisms. The main disadvantage of this design is the IPC overheads involved. Since the Linux kernel is not a microkernel or an RTOS, one of its main overheads comes from the implementation of the IPC code. The other disadvantage is that if the central server goes down, the whole phone stack is rendered unusable.

- The second kind implements several independent threads which are each responsible for a part of the phone stacks's functionality. For eg. There are separate threads for handling the wifi network, GSM telephony, audio etc. The advantage of having this design is that the overall architecture is simple to design and there is no single critical component in the architecture. The main disadvantage of this design is that each thread needs to implement the whole functionality in itself. eg. The SMS application will need the code for controlling the GSM engine. So it lacks the abstraction that is present in the first design. However, this kind of architecture is still present in old generation phones. The first implementation of the stack called FSO on the Openmoko Neo1973 had this design.

  The first type of architecture is more commonly present in today's smartphones. Userspace security solutions seem to fit in well with such designs. Programming language security features present in Java, SELinux labeling of processes and their access control and App Armor style application level sandboxing are some of the userspace security options.

5.2 Qtopia Model

Qtopia contains a critical component called QPE[39]. This is the main server that interfaces with the Operating system through device nodes and sockets for IPC communication. QPE is the first application to spawn when the stack executes. It opens all the necessary sockets and devices and then initiates the invocation of Message Server and Media Server. The Message Server controls the messaging and emailing functions of the stack. The Media Server controls the voice and audio related functions of the stack. There are several other applications like Qtmail, Games, Browser etc. that are invoked as plugins. These plugins are separate binaries that link with the QPE server at runtime. An application like Qtmail which implements the functionality of SMS, MMS, Email only contains the GUI code for displaying the text box and buttons. The graphical components like widgets are coded with the help of an API that is exported by the QPE server. So the QPE server implements the windowing manager as well. These widgets have mechanisms for triggering events. Eg. the messaging text box of the Qtmail plugin has a signal handler to receive the keys that are typed. The SEND button on the Qtmail screen also invokes another signal that results in constructing an Email or Message depending on the context. The following figure 5.2 shows the components of the Qtopia phone stack and the key interactions.

The signal itself and other data is transfered from QPE to the plugin via IPC mechanisms. In case of Qtopia, the communication channels are implemented as Sockets and Pipes. These are created by QPE during startup and then the connection at the other end is completed by the plugins when they are executed. Some of these sockets and pipes are named while others are unnamed. eg. one of the important communication sockets is called $/tmp/qtembedded-0/QtEmbedded-0$. This socket is opened by each plugin that connects with QPE to receive events. Although the name of the socket stays the same, the socket is identified by socket Inodes. In Linux, sockets are just like files and have a similar structure in the kernel. So, different applications communicating over the same named socket can still be identified uniquely in the kernel.

This form of IPC over sockets is implemented over AF_UNIX domain sockets. These are similar to the AF_INET socket family with the primary difference being in the machines involved in the communication. AF_UNIX sockets are used for inter process data transfers, whereas AF_INET is used for communication between remote processes running on remote hosts across a

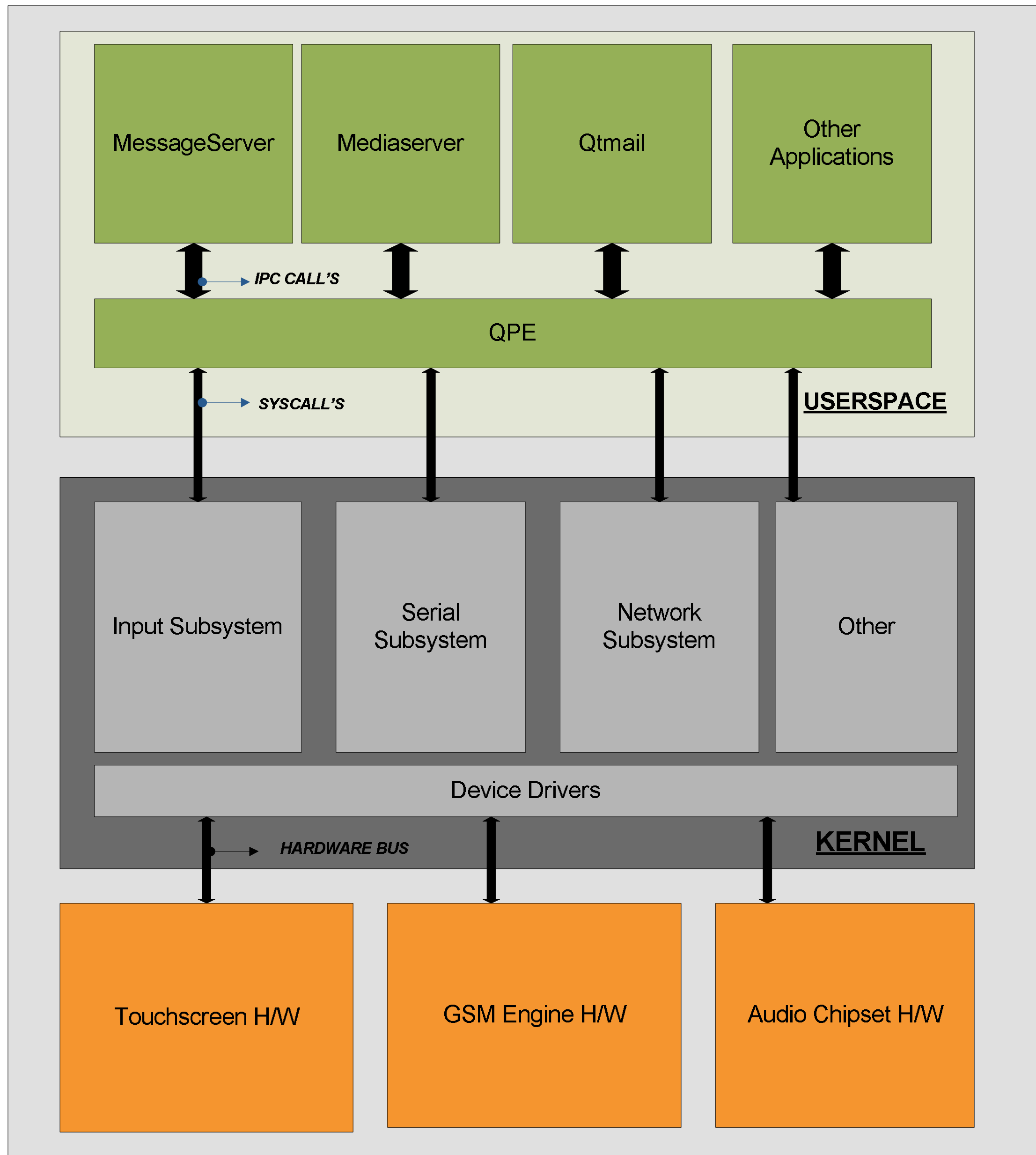

Figure 5.1. Qtopia Stack Diagram

network. The Linux implementation of AF_UNIX domain sockets is made such that a majority of the code is shared with the AF_INET code. The function operations (fops) performed over AF_UNIX sockets is separate so that the Send and Receive functionality for these sockets is identified uniquely. The actual data is transferred in a socket buffer (skb). This is a unit of network data transfer in the kernel. The skb data structure has all the information that is

required for identifying the sockets involved and the protocol used between the sockets.

5.3 Formalization

In order to represent the design formally we use the TLCK (Temporal Logic of Causal Knowledge) described by Bose et al. [21] The TLCK uses concurrency relations to describe the sequence of events in the system and the interactions in the Reference Monitor State Machine. The technical details of the reference monitor are explained in the next section.

The temporal events in the system are described using the following notations.

$\bigodot_t$ is an event true at time t.

$\triangle_t$ is an event true before time t.

$\square_t^{t-k}$ is an event true in the interval $[t-k, t]$.

The other relational operators such as $\wedge$ and $\vee$ etc. carry their usual meanings. Next we define some propositional variables.

- **KeyPressed(S):**

  Where, $S = \{s_1, s_2, s_3, .., s_n\}$ which is a set of $'n'$ scan code interrupts which we need to monitor. eg.$SEND, CALL, ENDCALL, RECORD, STOP$ etc. $KeyPressed(S)$ returns $TRUE$ if any of the monitored keys is pressed.

- **AuthenticateApp(T):**

  Where, $T = \{t_1, t_2, ..., t_n\}$ which is a set of $'n'$ applications that we need to authenticate. eg. $QPE, Qtmail, Mediaserver, Messageserver$ etc. $AuthenticateApp(T)$ returns $TRUE$ if the hash of the running application matches with a pre stored digest of that application. The hash is calculated only over the $TEXT$ segment of the application. A more detailed explanation is described in the next section.

- **IPC(T, Sockname):**

  IPC encapsulates both IPC read and write functions over the AF_UNIX socket. The name of the socket is defined by $Sockname$.

  Here we define some more notations for each IPC operation.

  - $A \xrightarrow{r} B$ denotes an IPC Read of application A from application B.
  - $A \xrightarrow{w} B$ denotes an IPC Write of application A to application B.

- **ParseATCMD(C):**

  Where, $C = \{AT{+}CMGS, AT{+}CHLD = 1\}$ is the set of AT commands to be monitored. $ParseATCMD(C)$ returns $TRUE$ when the reference monitor finds any of these commands in the *data* buffer that is passed to the GSM engine over the serial port.

5.3.1 Signatures

Now we can define signatures using the TLCK and the aforementioned notations. As a first example let us consider the signature for submitting an SMS.

The set $S = \{SEND\}$

The set $T = \{Qpe, Qtmail, Messageserver\}$

The set $C = \{AT + CMGS\}$

$Sockname = "/tmp/qtembedded - 0/QtEmbedded - 0"$

The truth table for $IPC(T, Sockname)$ is as follows:-

Variable	Value
$Qpe \xrightarrow{r} Qtmail$	T
$Qtmail \xrightarrow{r} Qpe$	T
$Qpe \xrightarrow{w} Qtmail$	T
$Qtmail \xrightarrow{w} Qpe$	T
$Qpe \xrightarrow{r} Msgserver$	T
$MsgServer \xrightarrow{r} Qpe$	T
$Qpe \xrightarrow{w} MsgServer$	T
$Msgserver \xrightarrow{w} Qpe$	T
IPC(T, Sockname)	**T**

Table 5.1. Messaging Truth Table

The IPC variable is $TRUE$ iff all the other variables are $TRUE$. Due to space constraints the $FALSE$ cases haven't been shown.

$$\bigodot_t(SubmitSMS) = \triangle_t(KeyPressed(S) \wedge AuthenticateApp(T))$$
$$\wedge(\Box_t^{t-k}(IPC(T, Sockname)) \wedge ParseATCMD(C))$$

This means that a real user pressed the $SEND$ key on the touchscreen/keypad, the applications in set T were authenticated by the Reference Monitor and there was an expected IPC transaction over the socket defined by $Sockname$ between these authenticated components within a time frame and the Reference Monitor received an $AT + CMGS$ command to the submit the SMS in the data buffer of the GSM serial port. The time frame for IPC can be customized depending upon the overheads of the IPC calls. When $SubmitSMS$ evaluates to $TRUE$, then the outgoing SMS is allowed, else denied. In a similar way, a signature for the audio attacks can be constructed as follows.

The set $S = \{CALL, ENDCALL\}$

The set $T = \{Qpe, Mediaserver\}$

The set $C = \{AT + CHLD = 1\}$

$Sockname = "/tmp/qt - embedded/valuespace_applayer"$

The truth table for IPC(T, Sockname) is as follows:-

Variable	Value
$Qpe \xrightarrow{w} Mediaserver$	T
$Mediaserver \xrightarrow{r} Qpe$	T
$Mediaserver \xrightarrow{w} Qpe$	T
$Qpe \xrightarrow{r} Mediaserver$	T
IPC(T, Sockname)	**T**

Table 5.2. Audio Call Truth Table

The IPC variable is $TRUE$ iff all the other variables are $TRUE$. Due to space constraints the $FALSE$ cases haven't been shown.

The signature for this is defined as :

$$\bigodot_t(AllowAudio) = \triangle_t(KeyPressed(S) \wedge AuthenticateApp(T))$$
$$\wedge(\Box_t^{t-k}(IPC(T, Sockname)))$$

Here the Reference Monitor looks for the $CALL$ scan code. When $AllowAudio$ evaluates to $TRUE$, then the Reference Monitor turns the microphone ON. At all other times, any command to alter the microphone state is denied. Since the Reference Monitor controls the microphone, it needs to know when to turn it OFF again. For this we have another signature.

$$\bigodot_t(DenyAudio) = \triangle_t(KeyPressed(S) \wedge AuthenticateApp(T))$$

$$\wedge(\Box_t^{t-k}(IPC(T, Sockname)) \wedge ParseATCMD(C))$$

Here the Reference Monitor looks for the $ENDCALL$ scan code. The truth table for this IPC operation in case of the Qtopia stack is the same as the case for $CALL$. In order to ensure that a call is being dropped or ended, the Reference Monitor parses the AT commands passed to the GSM engine. So, when it detects the command $AT + CHLD = 1$ after the IPC operations, it switches the microphone OFF.

Note that during the IPC transactions, the Reference Monitor also checks that the entities that are communicating are authentic. When $AuthenticateApp(T)$ returns $TRUE$, the reference monitor sets a bit in the process control block in order to avoid recalculating the hashes for every IPC call. This implies that $AuthenticateApp(T)$ is evaluated during the application load time.

5.4 Reference Monitor Design

This section describes the design of the Reference Monitor and how it functions to prevent the aforementioned attacks. The Reference Monitor is completely implemented as part of the kernel. The Reference Monitor consists of the following components:

5.4.1 Reference Monitor Interfaces

For the Reference Monitor to have complete mediation it needs to have hooks in all the security critical flows from the userspace application to the kernel that are relevant to the attack prevention.

- **Application startup:** The primary interface is the *exec* system call. This call invokes the process and sets up its memory maps for execution. At this point in time the reference monitor is able to authenticate the application. For the Qtopia stack the QPE, Qtmail, Messageserver and Mediaserver binaries are critical, since for any attack prevention these components are necessarily involved in IPC communication. For the authentication of these binaries, the reference monitor stores pre-computed md5 hashes of the $TEXT$ segments. The procedure for computing the $TEXT$ segment hashes is described in the Appendix section. When the binary runs, the Reference Monitor mediates the *exec* system call, to find out which binary is being loaded. If the binary name matches with either of the

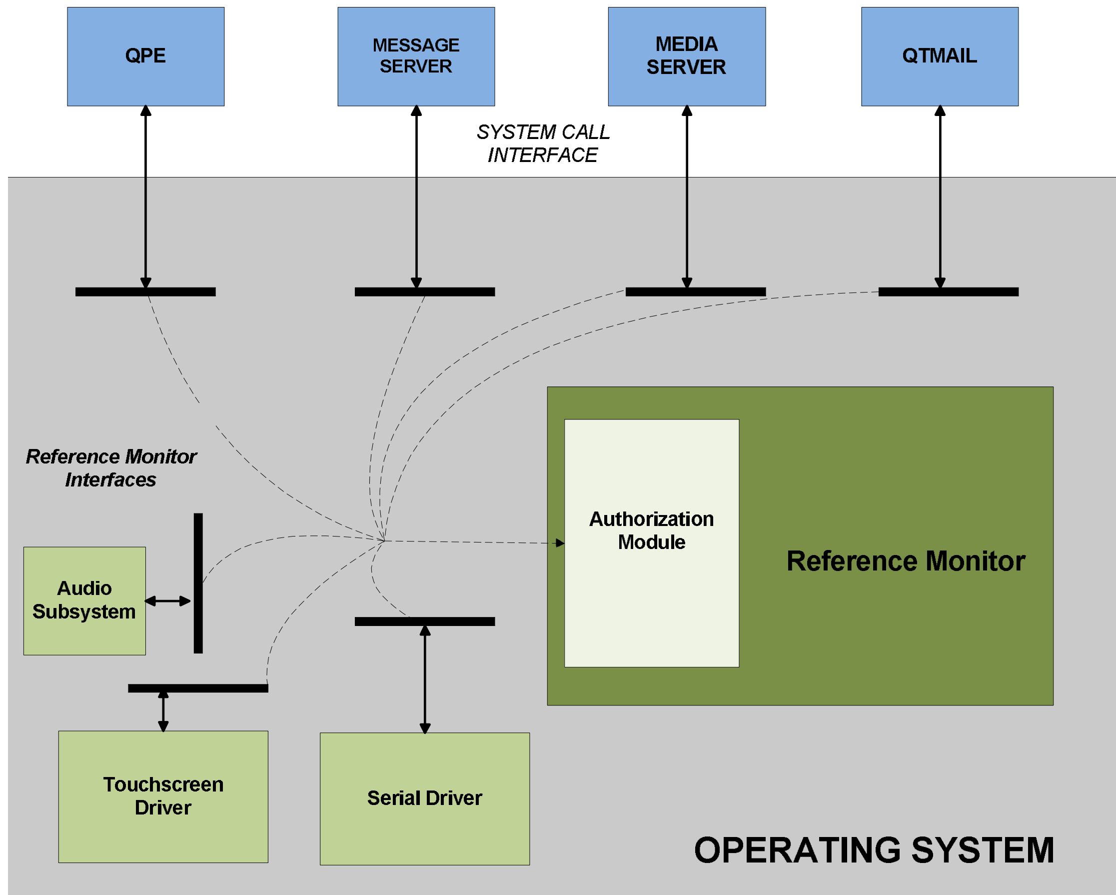

Figure 5.2. Reference Monitor

critical components, then it proceeds to generate a $TEXT$ segment hash of the about-to-run application and compares it with the respective stored copy. If the hashes match, then a permission bit is set in the PCB of the running application. This signifies that this application has been authenticated. This is useful while checking whether authentic applications are involved in the IPC communication as described in the following sections.

- **Input device scan code:** The next interface is the input device framework's send-to-user function. This function is responsible for sending the scan code from the touchscreen hardware to the userspace application that has registered to receive these events. At this point the reference monitor checks which key was pressed. If the scan code corresponds to some specific keys like the $SEND$, $CALL$ etc. key, then the Reference Monitor can decide the next steps. For any other key, the function simply returns control from the Reference

Monitor as it is irrelevant for the security purpose.

- **IPC Send and Receive:** The next interface is the AF_UNIX send and receive functionality. These are responsible for performing the actual IPC data transfer over sockets. As mentioned before, the sockets are identifiable by their unique inode numbers in the process context. The PCB of each process contains the inodes of all the sockets that are in use. When an application performs a send or receive operation on the socket, the CPU performs a mode switch and control is passed to the kernel via a system call. During this time the userspace application is paused, while the kernel functions on behalf of it. The kernel is able to identify which process invoked this system call. The Reference Monitor mediates both; send and receive system calls. The Reference Monitor is able to learn the socket inode numbers of the processes involved because it knows the name of the socket over which the processes communicate. For eg. the name of the socket */tmp/qtembedded-0/QtEmbedded-0* is pre-stored in the reference monitor. When one of the critical processes tries to send or receive over this socket, the Reference Monitor records the socket inode numbers after ensuring that the authenticated processes are involved in this transfer. During a pre-training run, the pattern of IPC communication amongst the critical components is observed by the Reference Monitor. During run time, the Reference Monitor then observes the currently ongoing IPC transfers between the processes to find a match in the IPC pattern. This pattern is described in the signatures as explained in the Formalization section.

- **GSM Serial port:** In order to communicate with the GSM engine, the applications in userspace have to use the serial port that is exported by the kernel serial subsystem. The read and write operations of this serial driver are also mediated by the reference monitor. The GSM port in the kernel driver is identified by a unique identifier. All data going through this port consists of the AT commands and the actual data that is transfered and received over-the-air. During the write system call to this driver, the Reference Monitor observes the data buffer to check for the $AT + CMGS$ command. If the reference monitor has concluded that malware is trying to send the message then it locks the GSM TX path thereby denying the SMS. If the buffer contains the $AT + CHLD = 1$ command, then reference monitor switches the microphone OFF. The action taken here depends on the attack that the Reference Monitor is trying to prevent and is described by the signatures

described earlier.

- **Audio configuration:** The audio controller has a multiplexer that is capable of routing audio data to and from multiple sources and sinks in the mobile device. eg. If the source of the audio is the MIC, then the sink could be the stereo speakers, or the headset, or even a bluetooth device. Likewise, the source of audio could also be a file with any of the sinks from the above. The configuration of these scenarios is done through an IOCTL command from userspace on the audio device node. The IOCTL function implementation in the kernel audio subsystem deciphers which kernel function to invoke depending upon the configuration of the scenario. The Reference Monitor mediates this IOCTL, since it is the entry point into the kernel from where the audio configurations are made possible. The significance of this interface is described in the Authorization Module functionality of the Reference Monitor.

5.4.2 Authorization Module

The Authorization Module of the Reference Monitor ensures that authenticated applications are involved in the IPC communication. It also decides whether to progress in the state machine towards preventing the attack vectors. The functions performed by this module are described below:

- When the plugins and servers begin to execute, they are checked for authenticity by comparing their $TEXT$ segment hashes as described earlier. This is to ensure that the critical components are not malware masquerading as the originals. In the Qtopia model, QPE, Qtmail, MessageServer and MediaServer are critical since these components are collectively responsible for the messaging and voice telephony functionality.

- By mediating all the key strokes' scan codes, the Reference Monitor checks if one of the monitored keys are pressed. Depending upon this the next stage is activated. All other key strokes are ignored.

- If the $SEND$ button is pressed, then the Reference Monitor looks for a pattern in the IPC communication between QPE, Qtmail and Message Server. These three components collaborate together in constructing and sending the message to the GSM engine. The

IPC pattern consists of AF_UNIX send and receive system calls on the named socket $/tmp/qtembedded-0/QtEmbedded-0$. For each send and receive from these components, a bit in a bit mask is flipped. This bitmask is a local variable to the reference monitor. A timer is started when QPE sends a message to Qtmail, since this is the beginning of the procedure to form an SMS. The timer is set to go off after 1 second. At the end of this second, the Reference Monitor checks for the final value of the bit mask. If this mask has an expected value then the Reference Monitor concludes that the $SEND$ key was pressed by the user for sending an SMS. Since the Reference Monitor has also checked that the inter communicating processes are authenticated, it is able to conclude that the user legitimately intended to send this SMS. After this the Reference Monitor opens up the TX path in the GSM port of the serial driver. Here a couple of things can arise:

- *The Reference Monitor incorrectly concludes that one of the monitored keys is pressed:* This may happen since the Openmoko device has a touchscreen, each button on the screen is defined by $[x, y]$ co-ordinates. Moreover a button spans a range of $[x, y]$ co-ordinates. These co-ordinates may have different semantics which depends on the application. eg. The co-ordinates for the $SEND$ button on the Qtmail application screen, may represent a different button, or action for the Mediaserver's applications like Mediaplayer. In this case, the ensuing IPC communication will differ from that of the signature and the Reference Monitor will not flip the bits in the bitmasks.
- *The Reference Monitor incorrectly concludes the key press events and starts the timers:* This may happen when the screen co-ordinates for the keys have different semantics as above and the IPC communication matches the expected pattern at the start. The match could occur because we start the timer upon the first IPC send or receive over the socket specified in the signature. In this case the timer will go off, but upon expiry the bitmask will have a different value at the end, because the later part of the IPC will not match as expected. The Reference Monitor reverts back to the initial state.

The basis for correctness of this approach is that when legitimate applications communicate over the named socket for constructing and sending an SMS, there is a unique pattern of IPC communication, so the Reference Monitor is able to guarantee that a message is on the way when it observes the pattern and gets the expected value in the bitmask.

For all other cases, when the Reference Monitor doesn't grant permission to send the SMS, the TX path in the serial driver has a hook where the Reference Monitor parses the outgoing AT commands. When there is no permission, the Reference Monitor garbles up the outgoing AT+CMGS command, thus denying the SMS.

Similarly for the audio attack vector, there is a unique IPC communication pattern between QPE and the Media Server over the named socket */tmp/qt-embedded/valuespace_applayer.* When the $CALL$ key is pressed, the Reference Monitor observes this IPC pattern and flips bit respectively. A similar timer as in the case for SMS, is triggered when QPE sends an IPC message to Media Server. Upon expiry of this timer, the value of the bitmask is compared. If it has the expected value, then the Reference Monitor concludes that a user intends to make a call and turns the microphone ON. During the duration of this call, the reference monitor locks the audio configuration path from the userspace. The IOCTL function in the audio subsystem has a hook, where the reference monitor denies any further configuration of the audio chip while the call is in progress. When the $END\ CALL$ key is pressed, the Reference Monitor switches the microphone OFF after monitoring for the expected IPC pattern and the occurrence of an $AT+CHLD$ command in the data buffer. In this manner the Reference Monitor is able to prevent malware from interfering with an ongoing call and also prevents it from capturing ambient sounds without the users knowledge.

5.5 Reference Monitor State Machine

The description of the Reference Monitor operations described above can be summarized with a State Machine as shown in figure 5.5.

5.5.1 States

- **State INT:** This is the Start state of the machine, where the Reference Monitor is just parsing the Input key events. The set of key scan codes to be monitored is dependent upon the signature. When either of these key presses are detected, the Reference Monitor transitions to the $IPC\ State$.

- **State IPC:** This is where the Reference Monitor begins to monitor the ensuing IPC com-

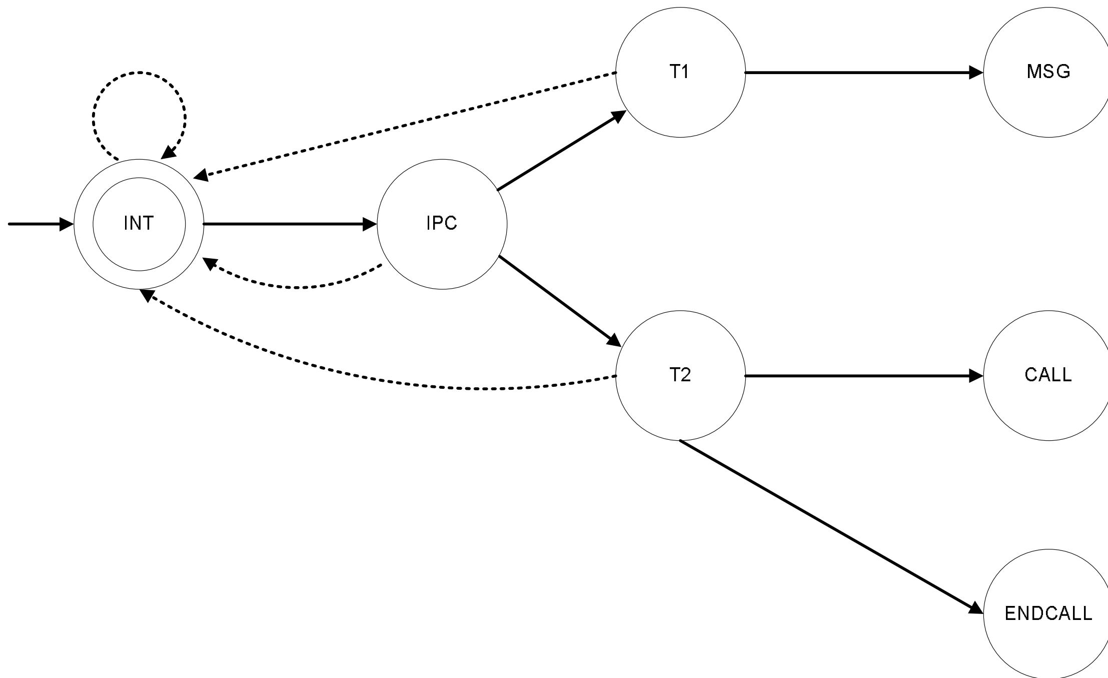

Figure 5.3. State Machine

munication between processes after a specific key press event. In this state, it also checks if the communication is being performed over a signature specified named socket and the communicating peer processes are authenticated. If they are authentic, then it proceeds to either the $T1$ or $T2$ $States$.

- **State T1:** Here, the Reference Monitor detects there was an IPC Send operation from Qpe to Qtmail ($Qpe \xrightarrow{w} Qtmail$). This is an indication that there could be a process to form an outgoing SMS. Then the SMS timer is set off. After this timer expires, the Reference Monitor checks the value of the bit mask which indicates how many IPC operations specified in the signature were performed. If the value of this mask is as expected and the Reference Monitor gets an $AT + CMGS$ command in the *data* buffer of the GSM serial driver, then the GSM TX is unlocked and the SMS is let though the GSM engine in $STATE\ MSG$.

- **State T2:** Here the Reference Monitor detects there was an IPC Send operation from QPE to Mediaserver ($Qpe \xrightarrow{w} Mediaserver$). This signifies that an outgoing GSM call could be in progress. The Call timer is set off. After this timer expires, the Reference Monitor checks the value of the bit mask as in the case of $State\ T1$ and if the expected

value is found, then checks if the $AT + CHLD = 1$ command is found in the *data* buffer of the GSM serial driver. If this command is found, then it signifies an end of call, so the Reference Monitor switches the microphone OFF, else it is turned ON.

5.5.2 Reverse Edges

The dashed back arrows in the State Machine figure 5.5 are explained here.

- **INT → INT:**

  This means the Reference Monitor has not encountered any key press events for the ones it is monitoring.

- **IPC → INT:**

  This could happen when :-

  - The Reference Monitor falsely recognized a key press event. This case is explained the Authorization Module section.
  - When unauthenticated entities tried to communicate over that socket.
  - When authenticated entities sent data over some other socket than the one specified in the signature.

- **T1 → INT:** Here the Reference Monitor started the SMS timer, but the ensuing IPC transactions didn't match with the signature.

- **T2 → INT:** Here the Reference Monitor started the Call timer, but the ensuing IPC transactions didn't match with the signature.

5.6 Reference Monitor Assessment

By definition [40] a Reference Monitor defines the necessary and sufficient properties of any system that securely enforces a mandatory protection system, consisting of the following guarantees.

- **Complete Mediation:** The system ensures that its access enforcement mechanism mediates all security sensitive operations.

- **Tamperproof:** The system ensures that its access enforcement mechanism, including its protection system, cannot be modified by untrusted processes.
- **Verifiable:** The access enforcement mechanism, including its protection system, "must be small enough to subject to analysis and tests, the completeness of which can be assured".

Using this definition, we assess the implementation using the assessment criteria question given in the book by Jaeger et al. [40]. The definitions in the book are designed around the Mandatory Access Control implementations. However, our implementation is slightly different since, we do not define any specific labels, objects or subjects. But, their questions can still be used as guiding points for analyses.

5.6.1 Complete Mediation

- *How does the Reference Monitor interface ensure that all security sensitive operations are mediated correctly ?*

  In this context, the security sensitive operations are those that are related to sending an SMS or making a GSM call. So, we need to make sure that the Reference Monitor mediates all the possible interfaces through which the hardware devices can be controlled. As mentioned in the Reference Monitor interfaces section, the GSM hardware, Audio hardware and Touchscreen hardware are exported as device nodes to userspace programs. Accessing these device nodes results in mode-switching into the kernel where the respective device drivers perform the functions as requested. The Reference Monitor implementation here, has hooks in all these drivers to mediate all accesses to the hardware and the data flow occurring at those instants. For device access, the only way applications can access them directly is through these device nodes. The kernel is assumed to be free of rootkits and the access to kernel memory is denied by disabling the */dev/kmem* node. This ensures that applications cannot alter the control flow of the kernel or use alternate means to access or interfere with device access. The program loader in the kernel is the only way for an application to begin execution. Therefore the *do_exec* system call is a mandatory call for an application to get a CPU timeshare. This ensures that the Reference Monitor is able to mediate all applications being loaded into the system.
- *Does the Reference Monitor interface mediate security sensitive operations on all system*

resources ?

Examples of system resources are, files, network interfaces, hardware interfaces and memory interfaces. In the current implementation, we do not mediate all system resources. Only the ones that are relevant to the attack prevention are mediated. eg. We do not mediate file system access or network access. However, by not mediating all system resources, the effectiveness of the Reference Monitor is not compromised, since all the security sensitive operations relevant to the attack prevention are mediated, as explained above. So, even if malware alters files on disk, the Reference Monitor Authorization Module is able to detect changes like masquerading. For network access, currently this implementation does not mediate access, but it will be necessary to add the required hooks to the network subsystem in order to prevent cross-service attacks where malware spreads through these interfaces.

5.6.2 Tamperproof

- *How does the system protect the Reference Monitor, including its protection system, from modification ?*

  Here, the protection system consists of the Reference Monitor code which contains the signatures and timers for preventing the attacks. In theory, the protection system consists of a *protection state* where subjects and objects are represented by lables and the state defines the operations on them. It also consists of a *labeling state* and a *transition state* which define the mapping of objects and subjects to labels and their transitions during the flow of control in the system. But for this implementation, we do not define any labels. So, the definition of protection system changes for our purposes. The critical parts of the Reference Monitor which contain the signatures, digests for application authentication and the timers for IPC monitoring are protected from modification since they are entirely in the kernel space and are inaccessible to any userspace applications.

- *Does the system's protection system protect the trusted computing base programs ?*

  In our case, the trusted computing base consists of only the operating system and the bootloader. The protection system does not protect the booloader, since it comes up only after the kernel boots.

5.6.3 Verifiability

- *What is the basis for the correctness of the system's trusted computing base ?* The bootloader and the operating system used for this implementation are completely open source. So, the correctness of their implementation is based on the extensive review process provided by the testers and kernel hackers around the world. Since only such reviewed code makes it to the final release of the kernel after a three week review window, the chances of coding bugs are minimal. The verifiability of the Reference Monitor code can be justified on the basis of its simplicity in terms of lines of code. As shown in the Appendix section, the total lines of code added to the kernel is only around 780. The compactness of this implementation makes it easier to review for errors and loopholes.

Chapter 6

Results

This section explains all the overheads added by the Reference Monitor interfaces. The instrumentation was performed completely in the kernel of the Openmoko device. The hardware of the device consists of a Samsung S3c2410 ARM9 core running at 266MHz, with 128MB SDRAM, 64MB NAND Flash. The Linux kernel version on the device was 2.6.24 with modifications to add the Reference Monitor core and its interface hooks. The Appendix section shows a $diff$ of the files that were affected.

The timing was gathered using the Linux Kernel Time API that is commonly used to calculate process startup times. The *Struct timespec* contains variables for *seconds* and *nanoseconds*. The API *do_posix_clock_monotonic_gettime*(&*timespec*), fills up the *timespec* structure with the values of seconds and nanoseconds since *epoch*. The pseudocode to profile a function call is as follows:

```
struct timespec delta, before, after;
do_posix_clock_monotonic_gettime(&before);
call_funtion();
do_posix_clock_monotonic_gettime(&after);
delta = timespec_sub(after, before);
printk( delta.tv_sec, (delta.tv_nsec / NANOSEC_PER_SEC /1000))
```

The *tv_nsec* can be divided appropriately to get values in μs or ms.

6.1 Application Text Segment Sizes

Since the Reference Monitor only calculates the hashes of the $TEXT$ segment of the critical applications, the following table lists the sizes as stored in the respective Process Control Blocks of the applications.

Application	**Size** (KB)
QPE	176
MediaServer	192
MessageServer	596
Qtmail	28

Table 6.1. Applications TEXT Segment Sizes

6.2 Application Load Time

The Reference Monitor affects the load time of only a select few applications. These applications for the Qtopia stack are QPE, Qtmail, MessageServer and Mediaserver. These applications collectively perform all the critical functionality of the telephony stack as explained earlier. All the other applications being loaded in the system are not considered by the Reference Monitor.

Since *do_execv* is the system call to load the application into main memory for the Linux kernel, the table shown below shows the overheads for this call with and without the Reference Monitor.

Application	**Time**	
	With RefMon (ms)	**Without Refmon** (ms)
QPE	374	2
MediaServer	210	78
MessageServer	561	66
Qtmail	40	10

Table 6.2. Application Load Overheads

The time taken for these applications to load through the Reference Monitor interface may seem very high. But this is because it scans through the whole $TEXT$ segment of the application, calculates an md5 hash and then compares this hash with a pre-stored digest that was computed in the training run phase. This scan may include a page table walk to fetch the respective pages into memory. However, this is only a one-time overhead, since the Reference Monitor sets a bit

in that applications PCB after comparing the hashes to signify whether the application has been authenticated. This bit is then checked during the IPC transactions thus avoiding re-calcuation of the hashes.

6.3 Training Run Time

For the training run, the Reference Monitor uses the same logic to calculate the hash of the running applications, but stores the hashes into a file, so that the trusted user may then statically link these digests with the kernel for run-time usages. The overheads for this procedure are shown below.

Application	**Time** (ms)
QPE	375
MediaServer	185
MessageServer	816
Qtmail	43

Table 6.3. Training Run Overheads

The overheads here include a file $WRITE$ operation to store the digest in the file on the flash device. After the training phase, the trusted user can then include these digests as static arrays into the Reference Monitor.

6.4 Input Event Overheads

The Reference Monitor parses the scan code of specific keys such as the *SEND, CALL, ENDCALL keys.* The overheads to check if these keys are pressed is shown in table 6.4.

Key	**Time** (μs)
SEND	1.0
CALL	1.1
ENDCALL	1.1

Table 6.4. Input Event Overheads

For the OpenMoko device, the touchscreen hardware produces screen co-ordinates which then map to scan codes. So, for every Key Press event, it generates several values as displayed below:

```
{type: 3, code: 0, value: 276}
{type: 3, code: 1, value: 222}
{type: 1, code: 330, value: 1}
{type: 3, code: 24, value: 1}
{type: 0, code: 0, value: 0}
{type: 3, code: 0, value: 267}
{type: 3, code: 1, value: 232}
{type: 0, code: 0, value: 0}
{type: 3, code: 0, value: 269}
```

Here, *code* : 0 signifies the $'X'$ co-ordinate and *code* : 1 signifies the $'Y'$ co-ordinate. The *value* contains the co-ordinate value. The *type* : 3 value is of relevance to the application that has registered to listen to these events. These prints are made by the kernel just before sending them to the userspace application.

Hence the Reference Monitor deciphers the keys according to the range of co-ordinates for each button.

6.5 IPC Overheads

The Reference Monitor monitors each IPC Send and Receive operation over the AF_UNIX sockets. However, it only mediates IPC operations over specific sockets and specific processes depending upon the signature.

With RefMon(μs)		
	Send	**Receive**
No Key Press	1.0	1.0
Not Authenticated	1.0	1.0
Authenticated	20	17

Table 6.5. IPC Overheads

Also, the time taken by the IPC operation depends upon the data being transfered between the two processes. This varies for every run. Hence the results shown in table 6.5 show the average time over 5 runs of sending an SMS and making a GSM call.

For each IPC operation, first the Reference Monitor checks to see if any of the monitored

keys was pressed. If not it simply returns, because that IPC operation was not triggered as a result of a hardware interrupt. Similarly, it then checks if the application that initiated the IPC operation was authenticated previously. If not, then it returns. This ends up preventing malware from sending an SMS or modifying the microphone state as explained by the signatures. If the application is authentic, then it checks which application sent or received data and on which socket. Accordingly it decides which timer to trigger as per the signature. The cases shown in table 6.5 show that the Reference Monitor takes on average $20\mu s$ more while trying to prevent malware activity and negligible overhead for all other cases.

Chapter 7

Conclusion

The framework described in this thesis shows a simple low overhead specification based intrusion prevention approach. The results section shows that the major overheads are only during the application startup. Since this is only a one-time overhead, it does not affect the usability of the device. The other overheads during the key scan code parsing, IPC monitoring and data buffer manipulation are minimal. The main motivation for using the specification based approach was due to the simplicity of the overall system as compared to the desktop. Since the mobile device always comes with one stack to implement the device functionality, the applications to be monitored for correctness are simple and few in number. This makes the specifications simple to describe and implement. Machine learning techniques used by anomaly detection approaches impose a greater overhead on the systems resources and have higher false positives. Misuse detection approaches also require a large amount of attack signatures and have a higher run-time monitoring overhead. Mandatory Access Control solutions require vast amounts of policies for enforcing correct program behavior and therefore have higher chance of weak security due to imprecise or insufficient policies. Our design has no false positives, has negligible overhead on the system and is flexible enough for adapting to various forms of telephony stacks.

Here we discuss some of the weaknesses and counter attacks of the described framework.

- We do not consider the integrity of the outgoing messages. So, malware could exploit a vulnerability of the messaging thread in userspace and piggy-back malicious data along with an outgoing message. In this case, the Reference Monitor will not be able to detect

such a tampered message, but still let it through since the message was initiated through the $SEND$ key interrupt and followed by the IPC pattern according to the signature. We think this is a challenging research problem using our framework. Some options to prevent this, would be to monitor each key scan code while forming the message in order to decipher what message was typed by the user. Then the Reference Monitor could parse the outgoing message data buffer to check if those letters are present. However, this would impose much higher overheads during parsing. Also, the data buffer may be encrypted by the message server, in which case, the Reference Monitor will not be able to decipher the contents. In such cases, we might need to trust the message server to protect the integrity of the message.

- If the application's IPC interaction is similar for more than one case, then we need to monitor for a stronger IPC pattern to uniquely identify the applications. eg. Consider that the signature for making a CALL and Recording audio contains the same IPC pattern. Since these two events are actually different, they should be uniquely identified by the Reference Monitor. They could have the same IPC pattern if they involve the same components in userspace to provide their functionality. eg. the CALL application and the Recording application could both use QPE and Mediaserver.In these cases, we need to monitor for other events caused in the system by these applications. eg. For the CALL application, we monitor for an outgoing $AT + CNUM$ command in the GSM serial data buffer to signify the callee's number. In other circumstances we could include other named sockets or pipes involved in the IPC.

- If the malware uses WiFi or BT for spreading, the Reference Monitor needs hooks in the Network stacks. The Openmoko device Neo1973 does not have a WiFi chip and the Qtopia Stack with the BT chip did not work during this research. However, we think, the current framework can be extended in the following way. For BT, it is necessary to scan for devices and then pair with the peer device before transferring a file. These system events could form the signature in the Reference Monitor. The trigger for setting up a BT transfer can be a key press event as is the case with other events. The key scan codes would be for instance, the $transfer\ file$ button in the file manager application. After this the Reference Monitor would look for the key presses and BT commands to scan for devices and pair with

devices. If all these events are triggered by key press interrupts in that particular order, the Reference Monitor can allow the BT transfer, else deny it.

For WiFi, the Reference Monitor could monitor for events that occur before sending data out of the network interface. However, this may work only for file transfer activity. If for instance, the user is browsing the internet over WiFi, it will be harder to define a signature. This is another challenging research problem.

Appendix A

Code Fragments

A.1 TEXT Segment Hashing

This snippet of code shows the API involved in finding the TEXT segment of the process and generating a md5 hash.

```
char *is_qpe = "/opt/Trolltech/Qtopia/bin/qpe";
char *is_mesg = "/opt/Trolltech/Qtopia/bin/messageserver";
char *is_media = "/opt/Trolltech/Qtopia/bin/mediaserver";
char *is_qtmail = "qtmail"; //This one comes from PRCTL
struct scatterlist sg[1];
..
struct crypto_hash *tfm;
struct hash_desc desc;
char qpe_digest[]    = {0x5b,0xe4,0x63,0x32,0x97,0x99,0x8,0xaa,
0xa6,0xa3,0x35,0x46,0xd9,0xb2,0x8f,0x70};
char msg_digest[]    = {0x33,0xe0,0xac,0x17,0xa7,0x20,0x1e,0x89,
0xcb,0xb3,0x96,0x8f,0x43,0x19,0x19,0xa3};
char qtmail_digest[] = {0xaf,0xe,0x72,0x67,0xcd,0xc8,0x12,0x76,
0x6f,0x56,0x81,0xd8,0x98,0x75,0x78,0x1d};
char media_digest[]  = {0x54,0x57,0x69,0xc3,0x8,0x7b,0x1b,0xdf,
```

```
0x13,0x68,0x14,0x4,0xd6,0x86,0x2e,0x56};
unsigned int start_code, end_code, code_diff;

//Get Application Name
if(strncmp(is_qpe, filename, 29) == 0) {
printk("%s\n", filename);
digest = qpe_digest;
qpe_pid = current->pid;
}
else if(strncmp(is_mesg, filename, 39) == 0) {
printk("%s\n", filename);
digest = msg_digest;
message_server_pid = current->pid;
}
else if(strncmp(is_qtmail, filename, 6) == 0) {
printk("%s\n", filename);
digest = qtmail_digest;
qtmail_pid = current->pid;
}
else if(strncmp(is_media, filename, 37) ==0) {
printk("%s\n", filename);
digest = media_digest;
mediaserver_pid = current->pid;
}
else
return;

//Begin Code page hash
start_code = current->mm->start_code;
end_code = current->mm->end_code;
code_diff = end_code - start_code;
```

```
kbuff = kmalloc(code_diff, GFP_KERNEL);

//Get TEXT Segment
len = access_process_vm(current, current->mm->start_code, kbuff,
code_diff, 0);
tfm = crypto_alloc_hash("md5", 0, CRYPTO_ALG_ASYNC);
desc.tfm = tfm;
desc.flags = 0;

result_buff = kmalloc(16, GFP_KERNEL);
memset(result_buff, 0, 16);

//Alloc one Scatter-Gather List
sg_init_one(&sg[0], kbuff, len);

//Compute Hash
ret = crypto_hash_digest(&desc, sg, len, result_buff);
hash_len = crypto_hash_digestsize(tfm);

if(memcmp(result_buff, digest,
hash_len) == 0) {
printk("Authenticated: %s: %d\n", current->comm,
current->pid);
current->is_authentic = 1;
}
else {
printk("Failed to authenticate: %s: %d\n", current->comm,
current->pid);
current->is_authentic = 0;
}
```

Appendix B

Files Affected

B.1 Git Diff

This shows the number of lines changed and files affected in the Linux kernel.

```
From bee1cc85dd25d7c68ac66a9390dbecdd33ca1a37 Mon Sep 17 00:00:00 2001
From: root <root@ashbert-laptop.(none)>
Date: Thu, 26 Feb 2009 11:25:33 -0500
Subject: [PATCH] Ashwin Chaugule
 avc114@cse.psu.edu

Kernel Refmon for Mobile Malware prevention
        modified:   drivers/input/evdev.c
        modified:   drivers/serial/s3c2410.c
        modified:   drivers/serial/serial_core.c
        modified:   fs/exec.c
        modified:   include/linux/sched.h
        modified:   include/sound/soc.h
        modified:   kernel/Makefile
        new file:   kernel/refmon.c
        modified:   kernel/sys.c
```

```
        modified:   net/unix/af_unix.c
---
 drivers/input/evdev.c         |    8 +
 drivers/serial/s3c2410.c      |    7 +
 drivers/serial/serial_core.c  |    7 +-
 fs/exec.c                     |   13 +
 include/linux/sched.h         |    4 +
 include/sound/soc.h           |    7 +-
 kernel/Makefile               |    2 +-
 kernel/refmon.c               |  718 ++++++++++++++++++++++++++++++++++++++++++
 kernel/sys.c                  |   11 +
 net/unix/af_unix.c            |   10 +
 10 files changed, 784 insertions(+), 3 deletions(-)
 create mode 100644 kernel/refmon.c
```

Bibliography

[1] Enck, W., S. Rueda, J. Schiffman, Y. Sreenivasan, L. St. Clair, T. Jaeger, and P. McDaniel (2007) “Protecting users from ”themselves”,” in *CSAW ’07: Proceedings of the 2007 ACM workshop on Computer security architecture*, ACM, New York, NY, USA, pp. 29–36.

[2] Enck, W., P. McDaniel, and T. Jaeger (2008) “PinUP: Pinning User Files to Known Applications,” in *ACSAC ’08: Proceedings of the 2008 Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, USA, pp. 55–64.

[3] Coker, R. (2007) “Multi-category security in SELinux in Fedora Core 5,” *Linux J.*, **2007**(156), p. 3.

[4] Gosling, A., “Google Brings 34 Vendors To Phone Consortium,” .
URL http://www.mobilised.com.au/content/view/1179/96/

[5] Nakamura, Y. “SELinux for Consumer Electric Devices,” in *Linux Symposium*.

[6] Vogel, B. and B. Steinke (2007) “Using SELinux security enforcement in Linux-based embedded devices,” in *MOBILWARE ’08: Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp. 1–5.

[7] Kidman, A., “Mobile viruses set to explode Mobile viruses set to explode Mobile viruses set to explode Mobile viruses set to explode,” .
URL http://www.zdnet.com.au/news/security/soa/Mobile-viruses-set-to-explode/

[8] Cheng, Z. *Mobile Malware: Threats and Prevention*, *Tech. rep.*, McAfee.

[9] Gostev, A., “Mobile Malware Evolution: An Overview,” .
URL http://www.viruslist.com/en/analysis?pubid=200119916

[10] Emm, D., “Mobile malware – new avenues,” .
URL http://www.sciencedirect.com/

[11] Sekar, R., A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou (2002) “Specification-based anomaly detection: a new approach for detecting network intrusions,” in *CCS ’02: Proceedings of the 9th ACM conference on Computer and communications security*, ACM, New York, NY, USA, pp. 265–274.

[12] Kim, H., J. Smith, and K. G. Shin (2008) "Detecting energy-greedy anomalies and mobile malware variants," in *MobiSys*, pp. 239–252.

[13] Racic, C., Ma (SecureComm 06) "Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery," .

[14] Kruegel, C., W. Robertson, and G. Vigna (2004) "Detecting Kernel-Level Rootkits Through Binary Analysis," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, USA, pp. 91–100.

[15] Witkowski, T., N. Blanc, D. Kroening, and G. Weissenbacher (2007) "Model checking concurrent linux device drivers," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, New York, NY, USA, pp. 501–504.

[16] Chen, S., J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer (2005) "Non-control-data attacks are realistic threats," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, USENIX Association, Berkeley, CA, USA, pp. 12–12.

[17] Daniel P. Bovet, M. C. (2005) *Understanding the Linux Kernel*, O'Reilly.

[18] Love, R. (2005) *Linux Kernel Programming*, Novell Press.

[19] Openmoko, "Openmoko : http://www.openmoko.org," .

[20] Qtopia, "Qt-Extended :http://qtopia.net/modules/devices/," .

[21] Bose, A., X. Hu, K. G. Shin, and T. Park (2008) "Behavioral detection of malware on mobile handsets," in *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, ACM, New York, NY, USA, pp. 225–238.

[22] Guo, Z., Wang "Smart-Phone Attacks and Defenses," in *ACM SIGCOMM HotNets*.

[23] Cheng, J., S. H. Wong, H. Yang, and S. Lu (2007) "SmartSiren: virus detection and alert for smartphones," in *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, ACM, New York, NY, USA, pp. 258–271.

[24] Traynor, P., W. Enck, P. McDaniel, and T. La Porta (2006) "Mitigating attacks on open functionality in SMS-capable cellular networks," in *MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking*, ACM, New York, NY, USA, pp. 182–193.

[25] Enck, W., P. Traynor, P. McDaniel, and T. La Porta (2005) "Exploiting open functionality in SMS-capable cellular networks," in *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, ACM, New York, NY, USA, pp. 393–404.

[26] (2008) "Enhancing Security of Linux-based Android Devices," in *in Proceedings of 15th International Linux Kongress*, Lehmann.

[27] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff (1996) "A sense of self for unix processes," in *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, pp. 120–128.

[28] Mulliner, C., G. Vigna, D. Dagon, and W. Lee (2006) "Using labeling to prevent cross-service attacks against smart phones," .

[29] Christodorescu, M., J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith (2005) *Malware Normalization*, *Tech. Rep. 1539*, University of Wisconsin, Madison, Wisconsin, USA.

[30] Ruitenbeek, E. V., T. Courtney, W. H. Sanders, and F. Stevens (2007) "Quantifying the Effectiveness of Mobile Phone Virus Response Mechanisms," in *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE Computer Society, Washington, DC, USA, pp. 790–800.

[31] Xie, L., H. Song, T. Jaeger, and S. Zhu (2008) "A systematic approach for cell-phone worm containment," in *WWW '08: Proceeding of the 17th international conference on World Wide Web*, ACM, New York, NY, USA, pp. 1083–1084.

[32] Desmet, L., W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe (2007) "A flexible security architecture to support third-party applications on mobile devices," in *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, ACM, New York, NY, USA, pp. 19–28.

[33] Qtopia, "Safe Execution Environment," .
URL `http://doc.trolltech.com/qtopia4.3/sxe.html`

[34] Venugopal, D. (2006) "An efficient signature representation and matching method for mobile devices," in *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet*, ACM, New York, NY, USA, p. 16.

[35] Schmidt, A.-D., F. Peters, F. Lamour, C. Scheel, S. A. Çamtepe, and S. Albayrak (2009) "Monitoring smartphones for anomaly detection," *Mob. Netw. Appl.*, **14**(1), pp. 92–106.

[36] Venugopal, D., G. Hu, and N. Roman (2006) "Intelligent virus detection on mobile devices," in *PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, ACM, New York, NY, USA, pp. 1–4.

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

[38] Xu, N., F. Zhang, Y. Luo, W. Jia, and J. Teng (2009) "Stealthy Video Capturer: A New Video-based Spyware in 3G Smartphones," in *WiSec'09*, ACM.

[39] Nokia *Qt-Extended-4.4 Whitepaper*, *Tech. rep.*, Nokia.

[40] Jaeger, T. (2008) *Operating System Security*, Morgan and Claypool Publishers.