SECURITY IMPROVEMENTS TO THE ANDROID KERNEL

DISSERTATION

submitted by

ASISH K S CB.EN.P2CYS13005

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

SECURITY IMPROVEMENTS TO THE ANDROID KERNEL

DISSERTATION

submitted by

ASISH K S CB.EN.P2CYS13005

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

Under the guidance of

Prof. Prabhaker Mateti
Associate Professor
Computer Science and Engineering
Wright State University
USA

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE - 641 112

BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled **"SECURITY IMPROVEMENTS TO THE ANDROID KERNEL"** submitted by **ASISH K S, Reg. No. CB.EN.P2CYS13005** in partial fulfillment of the requirements for the award of the **Degree of Master of Technology** in **CYBER SECURITY** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering.

Dr. Prabhaker Mateti
(Supervisor)

Dr. M. Sethumadhavan
(Professor and Head)

This dissertation was evaluated by us on .....

INTERNAL EXAMINER

EXTERNAL EXAMINER

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE -641 112

TIFAC-CORE IN CYBER SECURITY

DECLARATION

I, ASISH K S, (Reg. No: CB.EN.P2.CYS13005) hereby declare that this dissertation entitled **"SECURITY IMPROVEMENTS TO THE ANDROID KERNEL"** is a record of the original work done by me under the guidance of **Prof. Prabhaker Mateti**, Associate Professor, Wright State University, and this work has not formed the basis for the award of any degree / diploma / associateship / fellowship or a similar award, to any candidate in any University, to the best of my knowledge.

Place: Coimbatore

Date: Signature of the Student

COUNTERSIGNED

Dr. M. Sethumadhavan

Professor and Head, TIFAC-CORE in Cyber Security

ACKNOWLEDGEMENTS

At the very outset, I would like to give the first honors to **Amma, Mata Amritanandamayi Devi** who gave me the wisdom and knowledge to complete this dissertation under her shelter in this esteemed institution.

I express my gratitude to my guide, **Prof. Prabhaker Mateti**, Associate Professor, Computer Science and Engineering, Wright State University, USA, for his valuable suggestion and timely feedback during the course of this dissertation.

I would like to thank **Dr M. Sethumadhavan**, Professor and Head of the TIFAC-CORE in Cyber Security, for giving me useful suggestions and his constant encouragement and guidance throughout the progress of this dissertation.

I express my special thanks to my colleagues who were with me from the starting of the dissertation, for the interesting discussions and the ideas they shared. Thanks also go to my friends for sharing so many wonderful moments. They helped me not only enjoy my life, but also enjoy some of the hardness of this dissertation.

In particular, I would like to thank **Mr. Praveen K**, Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore and **Mrs. Jevitha K. P** Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for providing me with the advice, infrastructure and all the other faculties of TIFAC-CORE in Cyber Security and all other people who have helped me in many ways for the successful completion of this dissertation.

Contents

List of Figures

List of Tables

Abstract

Android, though seen mostly in smart phones and tablets, has the inherent capability of a fully functional operating system like Windows, Linux and iOS. As a result, all desktop and server Linux exploits are applicable also to Android. Note also that the Linux kernel within Android has been tweaked so that it runs smoothly on mobile devices. But this is not well modified at the level of system calls. There is not enough documentation on system calls and near-zero research in recent years. This paper examines the system calls of Android and proposes adding and freezing of system calls based on the "smart phone" usage.

Keywords. Android, Lollipop, freezing system calls, adding system calls, Kernel, Linux, sys-call-table.

Chapter 1

Introduction

Android is the most widely used mobile platform today even though it came into existence in late 2008. Openness is the property that made people use Android in a wider range. Android was designed with users in mind. Users are provided visibility into how applications work, and control over those applications. This openness property have created users study, analyse and experiment their speculations with a minimal set-up. This has led to users exploiting holes in Android and becoming attackers on a large scale.

Securing an open platform requires a robust security architecture and rigorous security programs. Android was designed with multi-layered security that provides the flexibility required for an open platform, while providing protection for all users of the platform. The openness property has made Android to be designed in a way to reduce the burden in developers. As a result, some security controls were reduced.

Android with the use of advanced hardware and software, the data collected by the applications are exposed through the platform to bring innovation and value to consumers. To protect that value, the platform must offer an application environment that ensures the security of users, data, applications, the device, and the network. Android was designed to both reduce the probability of these attacks and greatly limit the impact of the attack in the event it was successful.
Android has rapidly become the fastest-growing mobile OS. Android built on the contributions of the open-source Linux community contains more than 300 hardware, software, and carrier partners. This rich and open framework has made it a favourite for consumers and developers alike, driving strong growth in app consumption. Android users download more than 1.5 billion apps and games from

Google Play each month. With its partners, Android is continuously pushing the boundaries of hardware and software forward to bring new capabilities to users and developers.

1.1 Motivation

Android is the most widely used platform for smart phones as of 2013. Android provides support for a variety of file systems (e.g., `ntfs`), has a full network layer, has a full process management and virtual memory, and so on. In short, Android, though mostly seen in smart phones, has the capability of a full operating system. But, this also means that the attack surface is wide. Also, to be noted is that the typical Android user is likely to be computer illiterate, may not have used a computer system ever.

1.2 Problem Statement

Proper handling of system calls provides stronger security to Kernel. The system calls in Android must be analysed carefully for security vulnerabilities. System calls that are not required should be deleted. In this context, we focus on security improvements to Android that need to be made at layers below the Android Framework. In particular, we focus on improvements at the Linux kernel embedded within Android for ARM devices. We are working with the latest Android AOSP [Torvalds and Others (2015)] and propose some system calls to be added/ deleted/ and/or frozen aimed at improving kernel level security.

1.3 Contributions made

The thesis dynamically analyses all the `/system/bin` and `/system/xbin` binaries of Android Lollipop and statically analyses the AOSP sources and suggests removal of some system calls and freezing of a few. The thesis also provides a mechanism for freezing system calls in Android thereby adding additional security to the system. A new and effective method for debugging kernels is also introduced in this thesis. Nearly 95% of smart phones devices are built with ARM architecture, the rest

with Intel Atom. Hence, the thesis has a wide scope as the device is being used at a very large scale.

1.4 Organization

The remainder of the thesis is organised as follows: Chapter 2 provides literature survey and the background work undergone for for appreciating the rest of the thesis. Chapter 3 analyses system calls in standard Linux kernel based on two approaches. Chapter 4 describes how system call can be added to standard Linux. Chapter 5 gives the proposed modifications to Android kernel. Chapter 6 describes the dynamic analysis of the same. Chapter 7 describes the static analysis of system calls done in Android. Chapter 8 describes the proposal for freezing system calls. Chapter 9 proposes an efficient way of debugging kernels. Chapter 10 highlights the previous works done on system calls. Chapter 11 provides evaluation of the proposals done. Chapter 12 gives concludes the thesis.

Chapter 2

Background

This section gives the necessary background for the rest of the paper. This section summarises literature surveys on Android security and describes the system calls present in standard Linux kernel and Android kernel. It also gives an outline on system call categories and their representation in Android which helps in understanding the proposals. Since Android is heavily dependent on ARM architecture, we are focussing our work only on ARM CPU.

2.1 Introduction to System Calls

Android runs on top of a Linux kernel. Linux was developed in pre-mobile era. Features like SIM cards, accelerometer, gyroscope etc. specific to smart phones were absent.

2.1.1 System Calls

System calls acts as entry point to OS kernel. System calls are critical components in a computer system. They are the mechanism by which a user process is granted access to resources critical to the system. Services like handling hardware devices, process management etc which forms the integral functions of kernel are all managed by system calls. They contain highly privileged instructions by which we communicate with hardware. A system call isn't an ordinary function call. They differ from regular function calls as the code being called resides in the kernel and a special procedure is required to transfer control to the kernel. However, GNU

C library provides wrapper functions which prevents users from learning the low level implementation details of the kernel. Most of the Linux system calls come with wrapper functions so that they can be called easily.

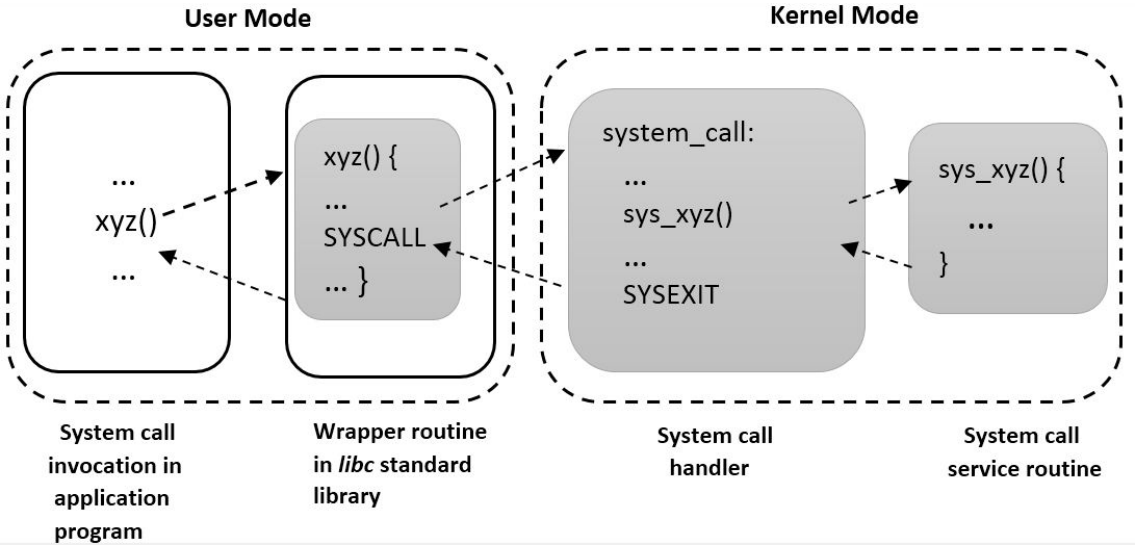

FIGURE 2.1: System Call Architecture

Figure 2.1 describes how system calls are invoked. The user program issues a system call which when executed passes control to its corresponding wrapper function. The wrapper function copies the system call number to eax register to the kernel and issues an interrupt which transfers execution to kernel mode. This is done with SYSENTER which produces a trap into the kernel and execute the system call. The system call handler will find the address of the corresponding system call service routine where the actual implementation resides and executes. After the execution of the routine system call handler will issue a SYSEXIT which changes the execution back to user mode and resumes program execution.

Android runs on top of Linux kernel. Linux was developed in pre-mobile era. Features like SIM cards, accelerometer, gyroscope etc. specific to smart phones were absent. In order to make it work on Android, Linux kernel was modified and tweaked. This kernel, even in the Lollipop version, lags behind the mainstream kernel found at `kernel.org` by several version numbers. As of 2015, Android uses kernel version 3.4.67 but main stream Linux kernel version is 4.1 as of June 2015. In Linux kernel 4.1[June 2015], there are 387 system calls [Torvalds and Others (2015)]. Though there has been considerable amount of modifications, it has not affected system calls definitions much. For a general overview of Linux architecture, see Bovet and Cesati (2005), and for Android, see Yaghmour (2013).

Detailed and accurate descriptions can only be deduced from actual source code files[1].

There may be ARM CPU dependencies in our work. We are not going to verify that our modifications will work on x86 based Android http://01.org, and http://www.android-x86.org/.

2.1.2 Categories of System Calls

The kernel contains system calls relating to processes (e.g., `fork` and `exec`), file systems (e.g., `open, close, read, write`), networking `socket, accept, bind, listen` and other resources that all services and user programs use. In Android, they have grouped system calls on the basis of theses classifications. Their way of representing system calls are found to be more vivid than in Linux. In Android, the file containing system calls are processed by a python script gensyscall.py by which all the system call stubs mentioned are automatically generated. The stubs generated can be called by a userland process to invoke the actual system call.

2.1.3 Understanding common System calls and related commands in Linux

1. Using strace command

   strace command is used for debugging system calls. It traces the execution of another program, listing any system call the program makes and any signal it receives.

   Eg: strace hostname provides a long listing where each line corresponds to a system call. They do not show ordinary function calls. The first line shows:

   execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0

   The first argument is the name of the program to run. The second argument corresponds to its argument list which consists of only a single element and the third is the environmental list. The standard C libraries are loaded from

[1] Files `arm/kernel/calls.S`, `arm/include/asm/unistd.h` and `arm64/include/asm/unistd.h` in https://github.com/torvalds/linux/blob/master/arch/ document the main line Linux system calls for ARM. The system call table for x86 architecture is in `linux-src/arch/x86/sys-calls/` directory. Files `SYSCALLS.TXT` and `include/unistd.h` in https://github.com/android/platform_bionic/blob/master/libc/ are Android specific.

a shared library file in the further lines. Lines at the end are system calls that actually make the program work.

uname({sys="Linux", node="myhostname", ...}) = 0

uname system call contains arguments sys and node which denotes operating system name and system's hostname.

write(1, "myhostname", 11) = 11

write system call produces output. Arguments 1 corresponds to standard output. The third arguement corresponds to the number of arguments to write and the return value is the number of characters that were actually written.

2. Using access command

   It is a system call used to check whether the calling process has access permission to a file. It takes 2 arguments. One is the path to the file and the other corressponds to permission checks to file. R_OK,W_OK,X_OK and F_OK are used to check read, write, execute and file access permissions. The access system call is returned with a 0 if the command successfully executes ie., the file has specified permissions.
   EACCES, EROFS, ENOENT corresponds to errors generated. They are defined under errno.h.
   Wrote a simple program using access system call to check whether a particular file is having access permissions or not. The full path to the file is given as argument.

3. Using fcntl system call

   It places a read lock or a write lock to a file. It corresponds to mutex locks. Fcntl consists of predefined structure called flock. A variable of flock structure is created, flushed with zeros. The l_type field denotes the lock to be placed. F_RDLCK is used for read lock and F_WRLCK for write lock. If a process holds a write lock to a file and another process tries to lock with write again, then fcntl blocks it until the first process releases it.
   Wrote a simple program using fcntl system call to lock a file using 2 terminals. One program acquires the lock while the other program waits for the release of the lock.

   NFS uses fcntl system call for locking its files by processes across multiple systems.

4. fsync
   System call that is used for writing important files to disk as soon as it is written to file. Takes a writable file descriptor as argument.

5. getrlimit, setrlimit System calls
   Allows processes to read and set limits to resources it can consume.

6. getrusage
   Retrieves process statistics from the kernel. It can be used to get statistics of the current process by using RUSAGE_SELF or of all child process forked by this process and its children using RUSAGE_CHILDERN.

2.2 System Call Table

System call table is an array of addresses of functions that provide the system calls. For ARM, the system call table is located in `arm/kernel/calls.S`. This table resides in a read-only protected virtual memory page, during run time. Adding/ deleting/ hooking system calls requires modifying this table.

2.3 Linux Kernel Module(LKM)

An LKM is a specially linked module (extension .ko) that can be inserted, at run time, to become part of the kernel address space. An LKM contains code that typically extends, and more rarely modifies, the functionality of the kernel. LKM can also be removed from the kernel without the need for rebooting. We use LKMs in this paper to freeze/redirect system calls by manipulate system call table.

2.4 System call for freezing

The thesis on "*Security Hardened Kernels For Linux Servers*" by Sunil Gadi explains about the adding a new system call that could freeze [Gadi (2004)] other system calls and itself once it has completed its operations.
There is a **sys_ni_syscall** already in the kernel, which is a no-op and always returns error code `-ENOSYS`, which stands for "the function is not implemented."

The system call number to be frozen is passed as an argument. The code for freeze system call is given below:

```
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}
asmlinkage int sys_freezesyscalls(int n)
{
    if (n < 0 || n > NR_syscalls)
       return -ENOSYS;
    if (!capable(CAP_SYS_ADMIN))
        return -EPERM;
    sys_call_table[n] = sys_ni_syscall;
    return 0;
}
```

`asmlinkage` is a macro which tells the compiler that it should check the arguments in the stack and not in registers. The freeze takes an argument n which corresponds to the system call number to be frozen. The argument is checked whether if it falls in the total number of system calls present and whether it has root privileges. If both theses conditions are achieved, then the values of the corresponding system call is made to point to `sys_ni_syscall` which when invoked returns an error message.

Chapter 3

System calls Analysis in Linux

As a proof of concept, the system calls in standard Linux kernel was analysed. The analysis was performed on the latest stable kernel available at that time. The system calls in kernel version 3.18.5 LTS was chosen for analysis. There are 320 system calls present in Linux kernel 3.18.5 LTS. Two approaches were followed for analysing.

3.1 Analysis based on functionality

The system calls are analysed based on the functionality it provides. The implementation of system calls were not taken into account. The system calls whose name does not denote the functionality and system calls which are old were taken and put to question whether this system call is necessary. The results are tabulated. Table 3.1 depicts unimplemented system calls.

TABLE 3.1: Table describing system calls that are not implemented

Number	Name
181	getpmsg
182	putpmsg
183	afs_ syscall
184	tuxcall
185	security
205	set_ thread_ area
211	get_ thread_ area
236	vserver

These system calls are not implemented and they are not being made to point to sys_ni_syscall.

Table 3.2 suggests removal of some system calls that removal from kernel. Their number, name, function it performs and reason for removal are shown here.

TABLE 3.2: Table describing system calls to be removed

Number	**Name**	**Function**	**Reason for suggesting removal**
7	poll	Manage file descriptors	Similar to select system call which takes an additional timeout argument.
19	readv	Reads specified amount of data from fd.	**preadv** also performs the same function with offset as an additional argument. We can give the offset as null which performs the same function as read.
20	writev	Writes specified amount of data to fd.	**pwritev** also performs the same function with offset as an additional argument. We can give the offset as null which performs the same function as write.
43	accept	Used for socket connection establishment	**accept4()** system call is available starting with Linux 2.6.28 with an additional flag options. But accept4 a non standard Linux extension.
61	wait4	Waits for process to change state	**wait3()** and **wait4()** functions are obsolete. They are replaced by waitid and waitpid. But they return additional info like resource usage. wait3() has been removed.
67	vfork	Creates new process and blocks parent	**vfork** is now an obsolete optimisation.
84	rmdir	Deletes directory if it is empty	We can use **rm** command with r option. **rm** uses unlink system call.

3.1.1 A review on poll and select system calls

The review was based on the USENIX conference Banga (1999). The system call poll takes the prototype

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Internally, `poll` maintains a structure for file descriptors and their events.

```
struct pollfd {
int fd;            // file descriptor
short events;     // requested events -- input
short revents;   // returned events -- output
};
```

Here, the argument fd is an array of file descriptors, an index corresponding each one of them. When the event requested by the process has not occurred for any file descriptors, then poll blocks until an event occurs. Timeout argument specifies the time (in ms) for blocking.
`Select` system call does the same function of waiting for file descriptors to become active with the occurrence of events. The system call select represents file descriptors as bitmaps. `Select` also maintains a structure like poll with more elements and bitmaps are are constructed using these before the select system call is called. This results in an inefficient way of programming when dealing with larger file descriptors. Poll and Select are slow in performance.
The `select` system call has a variant called `pselect6`. They differs in an argument in their prototypes. The system call `pselect6` is used with select when we are waiting for a signal or a file descriptor to become ready. It helps in preventing race conditions.[1]

Based on this, it is suggested to keep poll and its related ones like `ppoll`, `epoll_ctl`, `epoll_wait, epoll_create` system calls and remove `select` system call. Since `select` system call is removed, the existence of `pselect6` seems to be useless. Hence `pselect6` system call is also added to list of system calls to be removed.

Table 3.3 describes system calls that were analysed and found to be necessary.

3.2 Analysis using frequency of invocation

LTTng [Desnoyers and Dagenais (2006)] [Fournier et al. (2009)] provides both kernel level and user level tracing. We can trace system calls invoked using this. The logs are stored using session created for tracing. We also get the amount of time a particular event occurs. A service that runs LTTng was created and it was

[1]pselect6 had been falsly reported as bug in 2006 with medium severity. It was concluded as not a bug in 2007, https://bugzilla.redhat.com/show_bug.cgi?id=219676

TABLE 3.3: Table describing system calls that were analysed and found necessary

Number	Name	Function
24	sched_ yield	Releases processor from the running thread.
27	mincore	Checks whether pages are present in RAM so that disk access times are saved.
28	madvise	Gives advice to the kernel about handling paging for effective access.
99	sysinfo	Fetches information from /proc.Useful for obtaining network connection information.
139	sysfs	Useful for getting information about file systems. Used by /proc/filesystems.
275	splice	Transfers data between two file descriptors without copying between kernel and user address spaces.
277	sync_ file_ range	Synchronises file's data to disk from a given offset to the number of bytes provided as argument.
322	seccomp	Used for filtering system calls from the set passed to the application.

made to start at boot time at run levels 1-5. The system call traces produced from these run level were dumped to a unique location in the file system. Meaningful informations were extracted from the logs produced, making a clear picture of system calls and their frequency of execution.

With the successful execution of the command, the service was made to trace system calls at run-levels 1 and 2.[2] But the commands when executed provides unformatted output which becomes for a user to understand the semantics. Hence, a fairly simple converter known as Babeltrace was used. The converter takes the output of LTTng as input and formats them into meaningful data readable by the user. The output of Babeltrace converter was combined with multiple commands to produce output in table 3.4.

[2]Unlike standard Linux, in Ubuntu, the run-levels from 2 to 5 are considered to be having the same functionality.

2	name	=	sys_accept
9	name	=	sys_access
3	name	=	sys_brk
1	name	=	sys_clone
16	name	=	sys_close
2	name	=	sys_connect
28	name	=	sys_epoll_wait
1	name	=	sys_execve
1	name	=	sys_exit_group
43	name	=	sys_advise64
2	name	=	sys_fcntl
136	name	=	sys_futex
8	name	=	sys_getcwd
1	name	=	sys_getegid
1	name	=	sys_geteuid
1	name	=	sys_getrlimit
10	name	=	sys_gettid
2	name	=	sys_getuid
26	name	=	sys_inotify_add_watch
1886	name	=	sys_ioctl
23	name	=	sys_mmap
14	name	=	sys_mprotect
2	name	=	sys_munmap
1	name	=	sys_nanosleep
9	name	=	sys_newfstat
8	name	=	sys_open
1	name	=	sys_pipe
1408	name	=	sys_poll
696	name	=	sys_read
1537	name	=	sys_recvmsg
48	name	=	sys_rt_sigaction
321	name	=	sys_rt_sigprocmask
225	name	=	sys_select
140	name	=	sys_sendmsg
222	name	=	sys_settimer
2	name	=	sys_setpgid
1	name	=	sys_set_robust_list
2	name	=	sys_setsockopt
1	name	=	sys_set_tid_address
2	name	=	sys_shutdown
2	name	=	sys_socket
86	name	=	sys_splice
86	name	=	sys_sync_file_range
16	name	=	sys_unknown
2	name	=	sys_wait4
8	name	=	sys_waitid
860	name	=	sys_write
407	name	=	sys_writev

Table 3.4: Ouput of LTTng tracing syscalls in run-level 5.

Chapter 4

System calls Addition

4.1 Adding system call to Standard Linux Kernel 4.0.5

Linux has made a large leap in releasing stable kernel versions. The latest kernel version 4.0.5 was the latest stable one during the course of the project. A system call that adds two numbers was added to the kernel. The system call 'add' takes two integers arguments.
The following steps were followed.

1. Added an entry to system call table in kernel source (sys_add).
2. Added a prototype for 'add'.
3. Created a definition for the new system call.
4. Created an entry in the Makefile of the directory where the source is located.
5. Compiled, built and installed the new kernel.
6. The new system call was verified using a userspace program.
7. Kernel output was obtained by verifying the `dmesg` output.

The definition for add system call is given below:

```
asmlinkage long sys_add(int arg1, int arg2)
{
     return(arg1 + arg2);
}
```

The Linux source containing the 'add' system call was built using the conventional procedure.

4.2 Adding system call to Android kernel

Android kernel of goldfish(emulator) version 3.4 was taken as the test bed. The source was pulled from git source. It has similar structure of files as in Linux with some extra files like config specific to Android. The following steps were performed.

1. In arch/arm/include/asm/unistd.h, added new system call with syntax `#define __NR_add (__NR_SYSCALL_BASE+385)` where 385 is system call number for new system call 'add'.
2. In arch/arm/kernel/calls.S, added entry for new system call 'add' as `CALL(sys_add)`.
3. Added prototype for add system call in include/linux/syscalls.h as `asmlinkage long sys_add(int arg1, int arg2);`
4. Add entry to Makefile.

4.3 Building Android kernel from source

Desktop having 4 GB of RAM and 250 GB of hard disk with Intel Core2 Duo processor running Ubuntu 15.04 was chosen as the host platform for cross-compiling. Used arm-linux-androideabi as cross-compiler. The following steps were performed.

1. Added the path to cross-compiler to $PATH environment variable.
2. Set architecture to which the source should be compiled as ARM.

3. Set the device configuration of goldfish for compiling as goldfish_armv7_defconfig. Modern goldfish CPU has cortex-a8 which is not supported by old device configuration file. Hence, goldfish_armv7_goldfish was used for building.

The built took approximately 15 minutes. Proper building of kernel leads to bzImage file being generated at arch/arm/boot directory. The command `emulator` with kernel option was used to run the emulator with the newly built kernel. An application program was created for testing the system call added. The program was cross-compiled to make it work for ARM. The executable produced was pushed to the emulator using adb push. Figure 4.1 shows 'About phone' content of goldfish kernel.

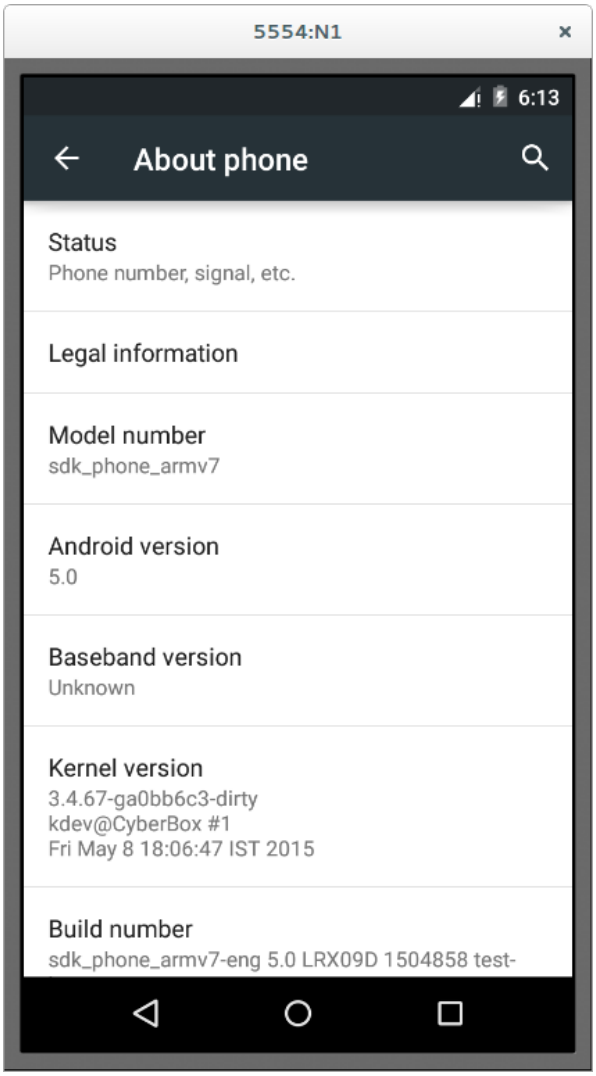

FIGURE 4.1: Screen shot of Android kernel build from source

Chapter 5

Proposed modifications to Android kernel

Android though aged only 6 years has taken the market of smart phones and is expanding to system level. Android has the capability of full-fledged OS like Linux, Windows and Mac. The openness of it to help the developers produce new apps has resulted in a large variety of attacks.Attackers are easily welcomed with source of internals which, when they explored found to be exploitable.

5.1 High level System Description

The project proposes to add a couple of new system calls, to delete a couple, and to freeze a few that taken together strengthen the security of the Linux kernel used within Android.

1. To analyse the system calls used in the latest Android Kernel.
2. To provide addition/removal of system calls based on the analysis.
3. To freeze system calls in Android kernel.
4. To provide a good solution for debugging the embedded kernel.

5.2 Detailed description of proposed system

5.2.1 Analysis of System calls in Android

Linux was developed in pre-mobile era. In order to make it work on Android, Linux Kernel is modified and tweaked. Currently, Android uses Kernel version 3.4 LTS but Linux has the version 4.1 which has been released in June 2015. For the kernel version 3.4.x which Android uses there are 180 files changed, 1285 insertions, 767 deletions. When an OS method like system calls are not considered for analysis, it has to be concluded that that the attack surface of Android is wide. Hence, the need for examining system calls is vital.

System calls are critical components in a computer system. They are the mechanism by which a user process is granted access to resources critical to the system. Services like handling hardware devices, process management etc. which forms the integral functions of kernel are all managed by system calls. The purpose of analyzing system calls in Android, is to make sure that each software component is invoking the exact list of system calls it is allowed to. There are two approaches viz. dynamic analysis and static analysis.

In dynamic analysis, the system calls that are invoked when a software component is executed are determined. A good diagnostic and debugging utility will be useful in providing this approach. This method determines system calls that are being invoked while a component is running. `Strace` and `ltrace` utilities list every system call invoked along with their duration and frequency of invocation. In Android, the system utilities are located in `/system/bin/` and `/system/xbin` directories. Some of them are soft links to toolbox. We perform an `strace` of files in these directories. Components like Zygote, binder, service manager etc which forms the core of the Android could also be traced for system calls using `strace`.

Static analysis aims to analyze the source code of all the software components for system calls. This provides a more reliable result than dynamically analysing system calls using debugging tools. In case of dynamic analysis, it is not always true that the output of `strace` is the only list of system calls invoked. Analysing system calls at the source code level offers to guarantee the actual list of system calls that a software component will invoke.
For instance, consider a utility/command that displays some network statistics. The utility/command requires the device to be connected to a network. If it

is having some network problems, it goes for invoking another system call that restarts the network. This will occur only if the network is down. During tracing of the command/utility, the system call for restarting the network will not show up when connected to network. Static analysis aims to list all the system calls that are present in the source no matter they are being called or not.
The paper proposes analysing system calls using pattern matching. The matching was performed on the Android kernel source using `grep` command. The option of searching multiple patterns, excluding some files irrelevant to the search, displaying line numbers of pattern matched etc. are the reasons for using the grep command.

5.2.2 Freezing of system calls in Android

Freezing of system calls is not the same as deletion, which is done at kernel compile/ build time. E.g., there are system calls that are required only during booting and for the initial configuration of the device. Consider the `mount` system call. Putting aside the need to insert/remove and hence mount/unmount removable SD cards, that savvy users want, it is more secure to disallow mounting/unmounting after boot time. Such system calls once they have finished their execution are no longer required.

Freezing system calls means that they are "non-existent" in the kernel system call table, at a certain chosen point in time, to be re-energized only on the next boot. A system call is frozen, not by deleting its definition, but by redirecting its entry point to a function that now returns the `ENOSYS` error value.

Enabling this requires the addition of a new system call `freeze` that freezes [Gadi (2004)] such system calls. The new system call freeze takes an integer value as an argument. The argument corresponds to the system call number of the system call to be frozen. The entry point to the system call definition is found using the system call number. This entry point is made to point to a function named sys_ni_syscall which points to NULL value thereby returning error value. System calls for freezing was done using Linux Kernel Modules(LKM).

5.2.3 Testing of New Kernels

The new Kernel which has been modified by adding and deleting system calls needs to verified for working and consistency. It is hard to check system calls every time by building the kernel and running them to test its working. Hence, a fairly elaborate test harness needs to be developed. There could errors encountered as a result of tainted kernel produced. In order to prevent them from damaging the machine, it would be effective to install the kernel in a virtual machine. A script was developed which takes the new generated kernel and an existing ISO as arguments. The script replaces the kernel present in the live ISO with the new modified kernel. The live ISO of Kubuntu 14.10 was taken as the testing framework. The procedure comprises of replacing `vmlinuz`, `initrd` image and library modules in `lib/modules` with the modified versions of respective items. The live ISO rebuilt was checked for working using VirtualBox. The entire procedure for building the bootable ISO from an existing one can be found at Mateti (2015).

Chapter 6

Dynamic analysis of system calls in Android

This method determines system calls that are being invoked while a component is running. `Strace` and `ltrace` utilities provide statistics on system calls invoked along with their duration and frequency of invocation. A command when used as an argument with `strace` will actually run the command. While running, it intercepts the execution to see any system calls that are being invoked. In Android, the system utilities are located in `/system/bin/` and `/system/xbin` directories. While running, `strace` intercepts the execution to see any system calls that are being invoked. We performed an `strace` of all commands in the `/system/bin` and `/system/xbin` directories. Android Framework components like Zygote, Binder, Service Manager, etc are also traced for system calls using `strace`.

6.1 Tracing Binaries

Table 6.1 shows `strace` output done on `ioctl` command. The %time denotes the percentage of time spent for invoking the system call. From the table it should be concluded that no time was spent for these system calls. As a result, the second and third column denoting seconds and microseconds spent are all zeros respectively. A total of 35 system calls were invoked when `ioctl` system call was invoked.

Consider another command used in Android known as `netcfg`. An `strace` on `netcfg` does not produce full output as the it calling the poll system call. This

%	seconds	usecs/call	calls	errors	syscall
0.00	0.0000000	0	1		set_tls
0.00	0.0000000	0	1		read
0.00	0.0000000	0	1		write
0.00	0.0000000	0	2		close
0.00	0.0000000	0	1		execve
0.00	0.0000000	0	11		sigaction
0.00	0.0000000	0	1		mprotect
0.00	0.0000000	0	1		writev
0.00	0.0000000	0	3		prctl
0.00	0.0000000	0	1		rt_sigreturn
0.00	0.0000000	0	1		sigalstack
0.00	0.0000000	0	2		mmap2
0.00	0.0000000	0	1	1	madvise
0.00	0.0000000	0	1		fcntl64
0.00	0.0000000	0	1		set_tid_address
0.00	0.0000000	0	1		clock_gettime
0.00	0.0000000	0	2		socket
0.00	0.0000000	0	2		connect
0.00	0.0000000	0	1	1	openat

TABLE 6.1: System calls `strace`-d in `ioctl`

means that `strace` is waiting for an argument. Hence, it should be concluded that `netcfg` system call always requires an argument. Each argument results in calling different sets of system calls although some may be common. Table 6.2 shows the output of strace when using the command,

```
strace -c netcfg lo down
```

Here, we can see that most of the time spent in executing the command is for the invocation of system call `fnctl64`. The system call `fnctl64` also accounts for the largest invoked system call(9583).

6.2 Tracing Zygote

The project traces the system calls which are the reason for the initialisation of Zygote. The program named app process is started by `init` as the service named zygote. It is present as an executable in `/system/bin/` directory. Table 6.3 shows the system calls found when `app_process` was traced. The first attribute **%time** denoting the percentage of time a system call is being invoked clearly says that a

%	seconds	usecs/call	calls	errors	syscall
100.0	0.026809	3	9583		fnctl64
0.00	0.0000000	0	1		execve
0.00	0.0000000	0	1		mprotect
0.00	0.0000000	0	1		sigalstack
0.00	0.0000000	0	1		mmap2
0.00	0.0000000	0	1	1	madvise
0.00	0.0000000	0	1		set_tid_address
0.00	0.0000000	0	1		openat
0.00	0.0000000	0	1		set_tls

TABLE 6.2: System calls `strace`-d in `netcfg`

%	seconds	usecs/call	calls	errors	syscall
91.24	0.004978	4978	1		writev
8.76	0.000478	478	1		mprotect
0.00	0.000000	0	1		read
0.00	0.000000	0	1		write
0.00	0.000000	0	2		close
0.00	0.000000	0	1		execve
0.00	0.000000	0	11		sigaction
0.00	0.000000	0	3		prctl
0.00	0.000000	0	1		rt_sigreturn
0.00	0.000000	0	1		signalstack
0.00	0.000000	0	2		mmap2
0.00	0.000000	0	1	1	madvise
0.00	0.000000	0	1		fnctl64
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		clock_gettime
0.00	0.000000	0	1		socket
0.00	0.000000	0	1		connect
0.00	0.000000	0	1	1	openat
0.00	0.000000	0	1		set_tls

TABLE 6.3: System calls `strace`-d in `app_process`

major amount of time was spent for invoking `writev` system call. These system calls are are always present every time we trace `app_process`. System calls other than these depend on the nature of the apps called.

Chapter 7

Static analysis of system calls in Android

Static analysis aims to list all the system calls that are present in the source no matter they are being called or not in a given run. The command grep which filters texts is used for analysing. The source of Android kernel has 10,370,303 lines of code. The codes specific to architecture are contained in arch directory and the rest are present in the root itself. The project aims in analysing architecture independent directories and the arm directory present in arch.

The source of goldfish kernel version 3.4 was used for performing static analysis of system calls. The system calls found in system calls table of ARM were stored in a file to match against system calls found in Android kernel source. Performed a trial acquisition of system calls in arm directory of Android goldfish kernel source. Output containing system calls obtained using grep on the ARM directory is shown below. It displays (to the right of the colon) the name of the system call matched along with the file name (at left) where this occurred. We can conclude that a major portion comes from calls.S which corresponds to the system call table of ARM.

```
arch/arm/kernel/sys_oabi-compat.c:sys_oabi_sendto
arch/arm/kernel/sys_oabi-compat.c:sys_oabi_sendmsg
arch/arm/kernel/sys_oabi-compat.c:sys_socketcall
arch/arm/kernel/calls.S:sys_restart_syscall
arch/arm/kernel/calls.S:sys_exit
arch/arm/kernel/calls.S:sys_fork_wrapper
```

```
arch/arm/kernel/calls.S:sys_read
arch/arm/kernel/calls.S:sys_write
arch/arm/kernel/calls.S:sys_open
arch/arm/kernel/calls.S:sys_close
arch/arm/kernel/calls.S:sys_ni_syscall
arch/arm/kernel/calls.S:sys_creat
arch/arm/kernel/calls.S:sys_link
arch/arm/kernel/calls.S:sys_unlink
arch/arm/kernel/calls.S:sys_execve_wrapper
arch/arm/kernel/calls.S:sys_chdir
arch/arm/kernel/calls.S:sys_mknod
arch/arm/kernel/calls.S:sys_chmod
arch/arm/kernel/calls.S:sys_lchown16
arch/arm/kernel/calls.S:sys_ni_syscall
arch/arm/kernel/calls.S:sys_ni_syscall
arch/arm/kernel/calls.S:sys_lseek
arch/arm/kernel/calls.S:sys_getpid
```

The command below when executed will provide a formatted output.

```
grep -R -o --exclude=calls.S -f pattern.txt AndroidSrc/arch/arm |
 cut -d ":" -f2 | sort | uniq -c
```

Table 7.1 shows the output obtained when the command is executed. It displays the name of the system calls invoked along with their frequency of invocation.

A script was created which displays the system calls present in Android source and their counts. The script takes the path to the Android source as the argument. The script also displays the total number of system calls that are used.[1] The source code of script is made available at Sahadevan (2015). The script requires the file containing the patterns or system calls to be present inside the root directory of the source. The script checks for in the following directories:

1. arch/arm
2. fs/

[1]The script checks for system calls in directories specific to ARM architecture. The script excludes the files `arch/arm/kernel/calls.S` and `include/linux/syscalls.h` which corresponds to system call table and prototype declaration respectively.

1	sys_arm_fadvise64_64
2	sys_bind
4	sys_clone
2	sys_connect
1	sys_creat
3	sys_epoll_ctl
2	sys_epoll_wait
5	sys_execve
2	sys_fcntl64
4	sys_fork
1	sys_fstat64
1	sys_fstatat64
2	sys_fstatfs
2	sys_fstatat64_wrapper
1	sys_ftruncate64
1	sys_lstat64
2	sys_mmap2
1	sys_newuname
1	sys_pread64
1	sys_pwrite64
1	sys_readahead
2	sys_rt_sigreturn_wrapper
1	sys_semop
2	sys_semtimedop
2	sys_sendmsg
2	sys_sendto
1	sys_sigaction
2	sys_sigalstack
2	sys_sigreturn_wrapper
1	sys_sigsuspend
2	sys_socket
1	sys_stat64
2	sys_statfs
2	sys_statfs_wrapper
1	sys_truncate64
4	sys_vfork

TABLE 7.1: Ouput of grep command with other filtering commands in Android source

3. drivers/
4. init/
5. kernel/
6. include/
7. net/
8. mm
9. ipc/
10. lib/
11. security
12. sound/
13. tools/

Table 7.2 shows the output was obtained.

The system calls from the arm architecture was taken as the pattern list. It has a total of 364 entries. The output produced contains 358 entries which means 358 different system calls were found out of 364 system calls in the pattern file. Hence, the rest 6 system calls does not occur in the search. The names of system calls which does not occur in the search were found using the command

```
grep -v -x -f out.txt pattern.txt
```

Count	System Call
3	sys_accept
8	sys_accept4
5	sys_access
5	sys_acct
2	sys_add_key
39	sys_close
8	sys_connect
11	sys_creat
3	sys_dup
2	sys_dup3
2	sys_epoll_create
37	sys_exit
2	sys_exit_group
2	sys_faccessat
4	sys_fallocate
10	sys_socketcall
11	sys_statfs
5	sys_umount
10	sys_unlink
19	sys_wait4
29	sys_write

TABLE 7.2: Ouput of script tracing syscalls in Android source and their number of occurrences.

Chapter 8

Freezing of system calls in Android

8.1 System call table

System call table is generally denoted as sys_call_table. The usage of sys_call_table is not strict though. The system call table, sys_call_table is actually a defined array. Its size is determined by NR_syscalls+1. NR_syscalls contains the total number of system calls that are present in the kernel. NR_syscalls accounts for both implemented and unimplemented system calls. To install our own system call, the entry point in the sys_call_table needs to be changed it points to the module function we are building. For that, address of system call table is necessary. System call table resides in a memory area after the kernel is built and installed. It resides in memory with readable only attribute set. In order to change the system call table, the page in which the system call table is located should be changed to writeable format. The address of sys_call_table is contained in Systems.map file. A regular pattern match in the file gives us the result.
The address of system call table can be found out using the command

```
grep -e "sys_call_table" Systems.map
```

To make changes into system call table, we must change the protection mode. There are two types of protection - Real and protected. In real mode, we could edit the system call table whereas it is free from protection in protected mode. Initially, when the kernel is compiled and built, the memory segment is in protected

mode. This protection mode is handled by a register called CR0(Control register). It has flags names like PE, WP for enabling and disabling memory protection.

8.2 Building LKM

A Linux Kernel Module(LKM) was built for ARM Android which manipulates the system call table thereby redirecting system calls. The unique property of the kernel modules to get inserted to and removed from the kernel without rebooting made us to create them. As a proof of concept, a LKM that could redirect the system calls was developed for x86-64 architecture [Mateti and Sahadevan (2015b)].

In case of x86-64 architecture, system call table are made writeable using lookup_address of type `pte_t`. In ARM, the function lookup_address located at `arch/arm/include/asm/xen/page.h` appears to be a bug returning NULL. A search for similar file in Android kernel source of goldfish led to file not being found on the source. Hence, it was concluded that the memory page does not need to be made writeable for modifying system call table entries.

8.3 System call Redirection

The source of goldfish kernel v3.4 was built with support for LKM enabled. A LKM was cross-compiled and built for ARM and inserted into the same kernel mentioned above. The source code of LKM is made available at Mateti and Sahadevan (2015a). The module redirects one system call to another.

Chapter 9

Boot ISO reconstruction

This chapter discusses the various procedures performed for replacing kernel`vmlinuz` and other relevant components.

9.1 Understanding mkinitramfs

Initramfs is a gzipped cpio achieve(lzma is also used often). At boot time, the kernel unpacks this achieve into RAM, mounts and uses it as initial root file system. This helps in mounting of the real root file system. klibc provides utilities to set up root. There are standard tools in Linux, `mkinitramfs` is one of them. It is provided by initramfs-tools package. This package depends on module-init-tools package which manages Linux kernel modules. Dracut [Hoyer (2013)] is an example.

The command `mkinitramfs` builds an initramfs image using the included scripts, necessary modules, udev, and utilities from klibc-utils or busybox. Usually runs in 2 modes.

1. Depmod
   Tries to find the necessary kernel modules or depending on the policy set in the config file, it will try to work out exactly which modules are needed to be in the system which might get wrong.

2. Most
   Includes all the disk drivers or net drivers. Useful when moving the entire root file system into a different hardware, it might still boot.

Performed a code walk through in `mkinitramfs` script.

1. Checks for busybox binary.
2. Executes user specified scripts in `/usr/share/initramfs-tools/scripts/functions` and `/usr/share/initramfs-tools/hook-functions`.
3. Prepare /lib/modules to be copied with the user specified kernel version.
   (a) Runs `depmod` if modules.dep does not exist. `depmod` kernel-version will use -a option by default which probes all modules.
   (b) Create temp directories using `mktemp` with name `mkinitramfs_XXXXXX` where XXXXXX is replaced by process number and random letters.
   (c) Populate temp dir with dirs contained in bin, conf/conf.d, etc, lib/modules, run, sbin, scripts, lib/modules/modules-directory. This includes modules which are not generated by `depmod` such as modules.builtin, modules.order etc.
   (d) Add modules from /usr/share/initramfs-tools/modules.d/. Checks for the 'dep', 'most' modes for adding modules.
   (e) Resolving hidden dependencies and copy init from /usr/share/initramfs-tools/init.
   (f) Execute `ldd` and copy-exec on binaries in /usr/share/initramfs-tools/bin dir to /sbin.
   (g) Generate module dependencies (uses `depmod -a -b`) to temp dir populated with the above procedures.
   (h) Copying ld.so.conf* and ldconfig tp update library search paths.
4. Extra functions like running optional scripts, caching the scripts , copying DSDT[4] to initramfs. These functions could be omitted.
5. cpio operation appears to be complicated.

9.2 Making initrd by hand

This section covers the major steps undergone for making initrd file without making use of mkinitrd/mkinitramfs script. The host machine contains Ubuntu 14.04 LTS OS. The procedure is followed based on the assumption that binaries present in /bin and /sbin directories are minimal.

1. Create a base directory for copying necessary files.
2. Create sub-directories with similar structure as that of root directory of the host file system namely, bin, conf, etc, run, sbin, usr, var.
3. Copy busybox binary to bin directory and make symbolic links to busybox.
4. Make softlink of init present in root directory to sbin directory.
5. Create directories dev, lib/modules, lib64, proc and sys.
6. Mount dev of type devtmps, proc of type proc , sys of type sysfs to /dev, /proc, /sys respectively.

Chapter 10

Related Work

This section summarizes the literature survey done for the project. The papers relevant to Android security in the recent years were taken.

10.1 Current Android Attacks

Most of the apps which we install in our Android devices are self-signed without employing any kind of central authority. The provision for the user to validate the authenticity of certificates is not available till date. The apps we are installing has the capability to register handlers for intents with priorities. There is no way of knowing what priorities are assigned to handlers at the time of installation and what its effects be [Vidas et al. (2011)]. There are some attacks that utilise the wide duration of patches cycles. The attacker gets quite a good amount of time to reverse engineer the patch before reaching the user. Attacks using Android Debug Bridge(ADB) are well known as attackers could get unprivileged interactive remote shell [Vidas et al. (2011)]. Another attack utilizing the recovery partition by loading malicious image without affecting user data has also been made possible. Even though there are quite a good number of apps that claim to "block malware, spyware and phishing apps, they require a rooted device to obtain the necessary permissions to perform these functions.

Moreover, a study on kernel source reveals that device drivers comprises of 70% of the code and accounts for largest error rates which ranges from three to seven times than rest of the kernel source and the new files introduced in Kernel has an

average rate about twice as high as the old files present can be found at Chou, Yang, Chelf, Hallem, and Engler (Chou et al.).

10.2 Improving Android security

In-spite of these attacks, Android has been improving its security both at the framework and kernel levels. Efforts to replace the existing DAC mechanism with flexible MAC has been made [Smalley and Craig (2013)]. Automatic security labelling of file systems particularly yaffs for newly created files was one big challenge.

In the framework level, the permissions requested and the API calls made by apps needs to be taken into account. Stowaway is one such tool that detects over-privilege in compiled Android apps. It determines the set of API calls that an app uses and then maps those API calls to permissions [Felt et al. (2011)]. The tool classifies the permissions requested by apps into Normal, Dangerous, and Signature/System permissions.

At the application level, Virtual ghosts [Criswell et al. (2014)] protects the apps from accessing its data and memory portions. The code and data partitions of the apps are stored in ghost memory which denies even the OS firm access. The pages used by ghost memory are non-writeable to prevent remapping of native code.

The paper Gadi (2004) provides an implementation of secure kernel for Linux servers. The paper suggested pruning of kernel thereby imparting additional security to the kernel. Freezing of system calls was introduced in this paper which disables those system calls which have already performed their task. The concept of freezing system calls proved effective since network attributes like IP address, iptables etc. are static.

The paper Mateti and Gadi (2003) analyses OWL, paged-PaX, seg-PaX patches and RSX module that prevents buffer overflow exploits. The paper highlights that fact that Linux is not using segmentation efficiently.

The behaviour of malware greatly depends on the invocation of system calls. These behaviours are analysed dynamically using CopperDroid [Reina et al. (2013)]. CopperDroid intercepts CPU privileges level transitions. It retrieves system calls whenever cpsr registers changes from supervisor mode to user mode.

Redesigning OS based on virtualization support in the CPU is now an active research area, e.g., Dune [Belay et al. (2012)], and Arrakis [Peter et al. (2014)]. Dune facilitates the application to have easy access to kernel resources like privilege modes, virtual memory registers, page tables, exceptions and system call vectors. IX [Belay et al. (2014)] aims to provide an operating system designed to break the 4-way trade-off between high throughput, low latency, strong protection and resource efficiency. It separates control planes (system configuration) and data planes (which controls networking stack and application logic). The ring 0 is divided into vmx-root and non-root which runs Linux kernel and IX respectively.

Linux Trace Toolkit Next Generation (LTTng) [Desnoyers and Dagenais (2006)] provides tracing of user and kernel space events.

There are, in fact, rootkits[1] which in addition to interposing system calls, can add new system calls to empty slots of system call table.

Many consider that a kernel built without LKMs, i.e., pre-link all the needed drivers, file systems, etc. to be far more secure compared to LKMs that have been exploited in the past. We use LKMs despite this. LKM insertions and removal are now being authenticated more vigorously [Woods (2001)].

[1] https://www.google.com/patents/US8955104

Chapter 11

Results and Discussion

11.1 Static Analysis of system calls in Android

Please note that the project considers system calls related to ARM arch only. Some of these system calls would appear in other architectures which are of least interest to us. Below are the 6 system calls which did not occur in the search.

1. `sys_time`
2. `sys_oldumount`
3. `sys_alarm`
4. `sys_old_readdir`
5. `sys_old_mmap`
6. `sys_sysfs`

The system calls except `sys_sysfs` are flagged as "Obsolete but used by libc4". The sys_sysfs system call in Linux is used to get system information but not used. Linux showed some paths to sys_fs. Tried a manual search on these directories. No result. Hence, it is concluded that sys_fs is also not used. Since these system calls are no longer used in Android, they could be removed. When looked into system call table, there are other obsolete system calls. This led to the understanding that Android is using some system calls that are obsolete. These obsolete system calls could give the attackers a chance to weaken the kernel. Hence they should be

put to analysis and appropriate measures have to be introduced to prevent them. They are

1. `sys_stime`
2. `sys_utime`
3. `sys_old_getrlimit`
4. `sys_old_select`
5. `sys_socketcall`
6. `sys_syscall`
7. `sys_ipc`

11.2 Freezing of system calls in Android

A system call which adds two integers(`sys_add`) was added to the kernel to test its working. An application program was used to check the working of `add` system call. The program was cross-compiled and pushed into the emulator. The application was executed to produce the result shown below.

```
System call invoked!
System call returned 60.
```

Here the value 60 corresponds to the result of addition of two integers passed as arguments. Kernel log after the application is executed is

```
<4> Add system call
```

The LKM was inserted using

```
insmod sysredirect.ko
```

A output of kernel logs obtained after inserting LKM is shown below.

```
<7>sys_call_table address = c00ef44
<7>Address of __NR_add = c001114c
<7>Inserting module
<7>New Address of __NR_add = c00393bc
```

When the application is executed once again, the output produced is

```
System call invoked!
System call returned -1.
```

The LKM was removed using

```
rmmod sysredirect.ko
```

A output of kernel logs obtained after removing LKM is shown below.

```
<7> Removing module
<7>Address of __NR_add = c001114c
```

From the logs, it is understood that the add system call has been redirected to another address. If the redirection address is that of sys_ni_syscall, then the system call has been frozen. The obsolete system calls which are used in Android can be put to analysis. They could be frozen using the LKM if their functionality falls only in a particular scope.

The presence of system calls added was confirmed by writing a user application program using that system call. The user program was cross compiled and its binary was pushed into emulator. The executed binary returns the expected results.

The total lines of code of kernel source without our modifications, nc = 10,373,553; after the mods, mc = 10,373,592. Only about 40 lines have changed. This shows that the proposed modifications have not made the kernel bulkier than it was before.

Even though the number of lines changed/ added is small, this is highly delicate work, requiring many hours of effort in designing, and even more hours of debugging.

Chapter 12

Conclusion and Future work

We would like to extend our work to support encryption at the kernel level. All the modern operating systems offers encryption in the *user space*. The addition of an atomic system call for encryption would prevent interception by user/kernel processes providing additional security. As of now, Linux supports only one password for a user. Its defined in `/etc/shadow`. Android offers users visit different websites as they do in Linux. A user will be having different passwords for different web sites. Storing passwords of a user based on the site he visited will reduce the effort of users. Acquiring information about sites and retrieving corresponding password can be done as a system call. Doing so would provide no way for an attacker to intercept the operation as they are atomic. Generalizing the existing mount related system calls to deal with cloud storage also is also highly desired. As a result, we could mount DropBox etc, as if they are local hard disk volumes, and available to *all* applications. The project wishes to replace the existing security mechanism with Capsicum capability, a lightweight OS capability and sandbox framework. One of the features of Capsicum is capability mode, which locks down access to global namespaces such as the file system hierarchy. In capability mode, `/proc` is thus inaccessible and so `fexecve(3)` doesn't work. Hence there is a need for a kernel-space alternative. The proposals implemented in this paper strengthen the Android kernel. The concept of freezing prevents the undesirable execution of system calls by malicious applications. The system calls frozen are not deleted but disabled, making them available in the next boot of the device.

The use of LKM makes the procedure simple as they could be inserted and removed without rebuilding or rebooting the kernel.

The proposal for analyzing system calls statically as described in this thesis has proven to be more reliable than dynamically analysing software components using tools. It highlights the fact that some obsolete system calls are still being used in Android. It also brings some system calls that are not used at all into the limelight.

Testing for new kernels described in the paper provides a convenient way for debugging kernels which reduces considerable amount of time. The script developed for this could be helpful for many kernel enthusiasts.

Chapter 13

Appendix

13.1 System calls analysis in Standard Linux using LTTng

Source code of script used as service

LTTng provides both kernel level and user level tracing. We can trace system calls invoked using this. The logs are stored using session created for tracing. We also get the amount of time a particular event occurs.

```
lttng created mysession            //creates a session
lttng enable-event --kernel        //trace all kernel events happening
lttng enable-event --kernel --syscall --all  //tracing all syscalls
lttng start session                // starts tracing
lttng stop
lttng view     // Displays whole list of events that have been
                  traced in the session.
```

A service was used which uses the LTTng service to get system call details at run-levels 1 and 2. The source code of service is as follows:

```
#! /bin/sh

case "$1" in
  start)
```

```
    echo "-----------------------------" >> /root/lttnglog
    echo "Starting script myservice ">> /root/lttnglog
    exec who -r >>/root/lttnglog
    exec /usr/bin/lttng create mysession123 >> /root/lttnglog
                                          2>> /root/lttnglog
    exec /usr/bin/lttng enable-event --kernel --syscall --all
     >> /root/lttnglog  2>> /root/lttnglog
    exec /usr/bin/lttng start >> /root/lttnglog 2>> /root/lttnglog
     ;;
   stop)
    echo "Stoping script myservice ">> /root/lttnglog
    exec who -r >>/root/lttnglog
    exec /usr/bin/lttng stop >> /root/lttnglog 2>> /root/lttnglog
    exec /usr/bin/lttng view >> /root/lttnglog.txt

    echo "-----------------------------" >> /root/lttnglog
   ;;
esac
```

The service was named "myservice.sh" and symbolic links were made to the corresponding directories in /etc./update-rc.d/. The script uses lttng service internally. Hence, to start "myservice", the following command was executed

```
    update-rc.d myservice start 21 1 2 3 4 5 . stop 19 0 6 .
```

The values '21' and '19' are given because the service which lttng [lttng-sessiond] works has to be started first in these run levels and killed only after killing myservice. For lttng-sessiond, the command

```
    update-rc.d lttng-sessiond start 20 1 2 3 4 5 . stop 20 0 6 .
```

was executed. Ouput of making symbolic links to directories is given below.

```
update -rc.d myservice start 21 1 2 3 4 5. stop 19 0 6.
 Adding system startup for /etc/init.d/myservice
    /etc/rc0.d/K19myservice  ->  ../init.d/myservice
    /etc/rc6.d/K19myservice  ->  ../init.d/myservice
```

```
/etc/rc1.d/K21myservice  ->  ../init.d/myservice
/etc/rc2.d/K21myservice  ->  ../init.d/myservice
/etc/rc3.d/K21myservice  ->  ../init.d/myservice
/etc/rc4.d/K21myservice  ->  ../init.d/myservice
/etc/rc5.d/K21myservice  ->  ../init.d/myservice
```

13.2 Static Analysis of system calls in Android

The details about the script created by which static analysis was efficiently perform for obtaining relevant informations is described in chapter7. The full output is shown below.

```
kdev@CyberBox:Androidkernel/goldfish-static-analysis$ ./systrace.sh
/media/kdev/Work1/Androidkernel/goldfish-static-analysis/
Tracing syscalls in Android Source and their number of occurences.
 The script requires the pattern file to be present in Android Src.
 The pattern file should be renamed as pattern.txt

   Count  System Call
-----------------------

      3 sys_accept
      8 sys_accept4
      5 sys_access
      5 sys_acct
      2 sys_add_key
      4 sys_adjtimex
      1 sys_arm_fadvise64_64
      2 sys_bdflush
      8 sys_bind
      2 sys_brk
      3 sys_capget
      4 sys_capset
      5 sys_chdir
      3 sys_chmod
      4 sys_chown
      1 sys_chown16
```

```
 6 sys_chroot
 5 sys_clock_adjtime
 5 sys_clock_getres
 5 sys_clock_gettime
 5 sys_clock_nanosleep
 5 sys_clock_settime
 2 sys_clone_wrapper
39 sys_close
 8 sys_connect
11 sys_creat
 3 sys_delete_module
 3 sys_dup
 1 sys_dup2
 2 sys_dup3
 2 sys_epoll_create
 3 sys_epoll_create1
 6 sys_epoll_ctl
 6 sys_epoll_pwait
 7 sys_epoll_wait
 2 sys_eventfd
 3 sys_eventfd2
 2 sys_execve_wrapper
37 sys_exit
 2 sys_exit_group
 2 sys_faccessat
 4 sys_fallocate
 2 sys_fanotify_init
 3 sys_fanotify_mark
 3 sys_fchdir
 2 sys_fchmod
 2 sys_fchmodat
 3 sys_fchown
 1 sys_fchown16
 1 sys_fchownat
 7 sys_fcntl
 7 sys_fcntl64
 4 sys_fdatasync
 1 sys_fgetxattr
```

```
 1 sys_flistxattr
 3 sys_flock
 2 sys_fork_wrapper
 1 sys_fremovexattr
 1 sys_fsetxattr
 2 sys_fstat64
 2 sys_fstatat64
10 sys_fstatfs
 2 sys_fstatfs64_wrapper
 5 sys_fsync
 6 sys_ftruncate
 4 sys_ftruncate64
 8 sys_futex
 5 sys_futimesat
 1 sys_getcpu
 2 sys_getcwd
 4 sys_getdents
 6 sys_getdents64
 1 sys_getegid
 1 sys_getegid16
 1 sys_geteuid
 1 sys_geteuid16
 1 sys_getgid
 1 sys_getgid16
 1 sys_getgroups
 1 sys_getgroups16
 4 sys_getitimer
 7 sys_get_mempolicy
 5 sys_getpeername
 2 sys_getpgid
 1 sys_getpgrp
 7 sys_getpid
 8 sys_getppid
 1 sys_getpriority
 1 sys_getresgid
 1 sys_getresgid16
 1 sys_getresuid
 1 sys_getresuid16
```

```
 4 sys_getrlimit
 7 sys_get_robust_list
 5 sys_getrusage
 1 sys_getsid
 5 sys_getsockname
10 sys_getsockopt
 1 sys_gettid
 5 sys_gettimeofday
 1 sys_getuid
 1 sys_getuid16
 1 sys_getxattr
 2 sys_init_module
 2 sys_inotify_add_watch
 2 sys_inotify_init
 4 sys_inotify_init1
 2 sys_inotify_rm_watch
 4 sys_io_cancel
42 sys_ioctl
 3 sys_io_destroy
 6 sys_io_getevents
 2 sys_ioprio_get
 2 sys_ioprio_set
 8 sys_io_setup
 6 sys_io_submit
 5 sys_ipc
10 sys_kexec_load
 7 sys_keyctl
 2 sys_kill
 3 sys_lchown
 1 sys_lchown16
 1 sys_lgetxattr
 5 sys_link
 3 sys_linkat
 5 sys_listen
 1 sys_listxattr
 1 sys_llistxattr
 1 sys_llseek
 4 sys_lookup_dcookie
```

```
 1 sys_lremovexattr
 7 sys_lseek
 1 sys_lsetxattr
 2 sys_lstat64
 4 sys_madvise
 7 sys_mbind
 3 sys_mincore
 5 sys_mkdir
 3 sys_mkdirat
 4 sys_mknod
 2 sys_mknodat
 2 sys_mlock
 2 sys_mlockall
 3 sys_mmap2
14 sys_mount
 7 sys_move_pages
 2 sys_mprotect
 7 sys_mq_getsetattr
 7 sys_mq_notify
 8 sys_mq_open
 8 sys_mq_timedreceive
 9 sys_mq_timedsend
 2 sys_mq_unlink
 2 sys_mremap
10 sys_msgctl
 8 sys_msgget
11 sys_msgrcv
10 sys_msgsnd
 6 sys_msync
 2 sys_munlock
 2 sys_munlockall
 4 sys_munmap
 3 sys_name_to_handle_at
 4 sys_nanosleep
10 sys_newfstat
 7 sys_newlstat
 6 sys_newstat
 4 sys_newuname
```

```
 4 sys_nice
13 sys_ni_syscall
 2 sys_oabi_bind
 2 sys_oabi_connect
 1 sys_oabi_epoll_ctl
 1 sys_oabi_epoll_wait
 1 sys_oabi_fcntl64
 1 sys_oabi_fstat64
 1 sys_oabi_fstatat64
 2 sys_oabi_ftruncate64
 1 sys_oabi_ipc
 1 sys_oabi_lstat64
 2 sys_oabi_pread64
 2 sys_oabi_pwrite64
 2 sys_oabi_readahead
 1 sys_oabi_semop
 4 sys_oabi_semtimedop
 2 sys_oabi_sendmsg
 2 sys_oabi_sendto
 1 sys_oabi_socketcall
 1 sys_oabi_stat64
 2 sys_oabi_truncate64
 2 sys_old_getrlimit
 2 sys_old_select
59 sys_open
 5 sys_openat
 8 sys_open_by_handle_at
 1 sys_pause
 1 sys_pciconfig_iobase
 1 sys_pciconfig_read
 1 sys_pciconfig_write
18 sys_perf_event_open
 1 sys_personality
 2 sys_pipe
 2 sys_pipe2
 1 sys_pivot_root
19 sys_poll
 4 sys_ppoll
```

```
 3 sys_prctl
 4 sys_pread64
 6 sys_preadv
 1 sys_prlimit64
 6 sys_process_vm_readv
 6 sys_process_vm_writev
 4 sys_pselect6
 5 sys_ptrace
 4 sys_pwrite64
 6 sys_pwritev
 6 sys_quotactl
36 sys_read
 5 sys_readahead
 1 sys_readlink
 2 sys_readlinkat
 4 sys_readv
 6 sys_reboot
10 sys_recv
11 sys_recvfrom
17 sys_recvmmsg
18 sys_recvmsg
 4 sys_remap_file_pages
 1 sys_removexattr
 2 sys_rename
 2 sys_renameat
 2 sys_request_key
 3 sys_restart_syscall
 3 sys_rmdir
 3 sys_rt_sigaction
 2 sys_rt_sigpending
 4 sys_rt_sigprocmask
 3 sys_rt_sigqueueinfo
 2 sys_rt_sigreturn_wrapper
 5 sys_rt_sigsuspend
 5 sys_rt_sigtimedwait
 4 sys_rt_tgsigqueueinfo
 5 sys_sched_getaffinity
 2 sys_sched_getparam
```

```
 2 sys_sched_get_priority_max
 2 sys_sched_get_priority_min
 2 sys_sched_getscheduler
 3 sys_sched_rr_get_interval
 5 sys_sched_setaffinity
 3 sys_sched_setparam
 2 sys_sched_setscheduler
 4 sys_sched_yield
 2 sys_seccomp
16 sys_select
13 sys_semctl
 8 sys_semget
 3 sys_semop
12 sys_semtimedop
 6 sys_send
 1 sys_sendfile
 1 sys_sendfile64
14 sys_sendmmsg
20 sys_sendmsg
10 sys_sendto
 1 sys_setdomainname
 2 sys_setfsgid
 1 sys_setfsgid16
 2 sys_setfsuid
 1 sys_setfsuid16
 3 sys_setgid
 1 sys_setgid16
 1 sys_setgroups
 1 sys_setgroups16
 1 sys_sethostname
 5 sys_setitimer
 7 sys_set_mempolicy
 1 sys_setns
 2 sys_setpgid
 3 sys_setpriority
 2 sys_setregid
 1 sys_setregid16
 2 sys_setresgid
```

```
 1 sys_setresgid16
 2 sys_setresuid
 1 sys_setresuid16
 2 sys_setreuid
 1 sys_setreuid16
 4 sys_setrlimit
10 sys_set_robust_list
 3 sys_setsid
11 sys_setsockopt
 1 sys_set_tid_address
11 sys_settimeofday
 2 sys_setuid
 1 sys_setuid16
 1 sys_setxattr
 8 sys_shmat
12 sys_shmctl
 3 sys_shmdt
 8 sys_shmget
23 sys_shutdown
 2 sys_sigaction
 2 sys_sigaltstack_wrapper
 6 sys_signalfd
 9 sys_signalfd4
 3 sys_sigpending
 6 sys_sigprocmask
 2 sys_sigreturn_wrapper
 1 sys_sigsuspend
 5 sys_socket
10 sys_socketcall
 5 sys_socketpair
 1 sys_splice
 3 sys_stat64
11 sys_statfs
 2 sys_statfs64_wrapper
 3 sys_stime
 7 sys_swapoff
 5 sys_swapon
 2 sys_symlink
```

```
 2 sys_symlinkat
19 sys_sync
 3 sys_sync_file_range2
 1 sys_syncfs
 6 sys_sysctl
 6 sys_sysinfo
 3 sys_syslog
 1 sys_tee
 2 sys_tgkill
 7 sys_timer_create
 1 sys_timer_delete
 2 sys_timerfd_create
 7 sys_timerfd_gettime
 7 sys_timerfd_settime
 1 sys_timer_getoverrun
 5 sys_timer_gettime
 6 sys_timer_settime
 8 sys_times
 2 sys_tkill
 5 sys_truncate
 4 sys_truncate64
 1 sys_umask
 5 sys_umount
10 sys_unlink
 1 sys_unlinkat
 4 sys_unshare
 2 sys_uselib
 4 sys_ustat
 5 sys_utime
 4 sys_utimensat
 6 sys_utimes
 2 sys_vfork_wrapper
 1 sys_vhangup
 5 sys_vmsplice
19 sys_wait4
 5 sys_waitid
29 sys_write
 5 sys_writev
```

```
Total number of different system calls invoked:
358
```

Bibliography

Banga, G. 1999. The problem with select(). https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/banga/banga_html/node3.html.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. 2012. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 335–348. https://www.usenix.org/system/files/conference/osdi12/osdi12-final-117.pdf.

Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. 2014. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14),(Broomfield, CO)*. 49–65. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

Bovet, D. P. and Cesati, M. 2005. *Understanding the Linux Kernel*. O'Reilly Media, Inc.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors.

Criswell, J., Dautenhahn, N., and Adve, V. 2014. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 81–96.

Desnoyers, M. and Dagenais, M. 2006. Low disturbance embedded system tracing with linux trace toolkit next generation. In *ELC (Embedded Linux Conference)*. Vol. 2006.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. 2011. Android Permissions Demystified. In *18th ACM Conference on Computer and Communications Security*. CCS '11. ACM, New York, NY, USA, 627–638.

Fournier, P.-M., Desnoyers, M., and Dagenais, M. R. 2009. Combined tracing of the kernel and applications with lttng. In *Proceedings of the 2009 linux symposium*.

Gadi, S. S. 2004. Security Hardened Kernels for linux Servers. M.S. thesis, Wright State University. Advisor: Prabhaker Mateti.

Hoyer, H. 2013. *Dracut Revision*. redhat.com. https://www.kernel.org/pub/linux/utils/boot/dracut/dracut.html.

Mateti, P. 2015. Boot iso reconstruction. http://103.5.112.91/pmateti/GradStudents/Asish/our-kernel-iso.html.

Mateti, P. and Gadi, S. S. 2003. Prevention of buffer overflow exploits in ia-32 based linux. In *Security and Management*. 378–384.

Mateti, P. and Sahadevan, A. K. 2015a. System calls redirection in android. https://github.com/asishsahadev/armsysredirect.

Mateti, P. and Sahadevan, A. K. 2015b. System calls redirection in x86-64 systems. http://103.5.112.91/pmateti/GradStudents/Asish/sysredirect.c.

Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. 2014. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

Reina, A., Fattori, A., and Cavallaro, L. 2013. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*.

Sahadevan, A. K. 2015. Tracing syscalls in android source. https://github.com/asishsahadev/systrace/blob/master/systrace.sh.

Smalley, S. and Craig, R. 2013. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*.

Torvalds, L. and Others, M. 2015. *The Linux Kernel*. http://www.kernel.org.

Vidas, T., Votipka, D., and Christin, N. 2011. All your droid are belong to us: A survey of current android attacks. In *WOOT*. 81–90.

Woods, G. A. 2001. Lkm insecurity. http://seclists.org/incidents/2001/Jan/33.

Yaghmour, K. 2013. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., Sebastopol, CA 95472.

List of publications

Asish K S, Prabhaker Mateti.2015. Security Improvements to the Android kernel. In *Eleventh International Conference on Information Systems Security*, Springer Verlag series of Lecture Notes in Computer Science.