A Security Enhancement for Android's Root File System*

Anh-Duy Vu, Jae-I1 Han**, Young Man Kim
{anhduyvu, jhan, ymkim}@kookmin.ac.kr

Abstract: Recently it has been a focus on protection of mobile devices such as smartphones, tablet and etc. Using Mandatory Access Control (MAC) is a promising way, however this approach requires that the root file system of a device must support extended attributes (xattr). This paper presents the design of YAFFS (the root file system of Android-powered devices) and describes how to implement extended attributes for this kind of file system. This supplement accelerates not only the utilization of existing MAC models such as SELinux, Apparmor, Tomoyo, Smack into the system, but also the development of pristine and dedicated models. To evaluate our implementation, we utilized the Bonnie++ benchmark tool and executed each benchmark 10 times. The results shows the trade-off between the benefits and risks when applying the additional feature to the file system such as decrement of CPU consuming when writing blocks but speed slowdown when reading blocks.

Key words: YAFFS, extended attributes (xattr), Linux Virtual File System Switch (VFS)

1 Introduction

Android is a Linux-based and comprehensive open-source operating system designed for mobile devices such as smart phones, tablets, music players, set-top-boxes, and etc. After the first introduction in 2007, Android has been developed by the Open Handset Alliance led by Google and grasped the attention of many mobile device manufacturers, service providers and application developers. Its openness gives boom to Android application market. However, Android Market lacks of dedicated team to analyze the application code to decide their trustworthiness while Apple App Store and Window Phone Marketplace do. Although Android Security group have deployed a malware s calmer, Bouncer 1, for Android Market, it hardly ensures that users are completely free from all types of attack of malicious applications.

We have extended Android security framework with a novel MAC model realizing a compact Domain and Type Enforcement (DTE) 2. In this paper, we present a part of this project which is responsibility on enhancing the Android Root file system by implementing extended attributes.

The remainder of this paper is organized as follows. Section 2 presents the overview of Linux Virtual File System Switch (VFS): its role, data structures, implementation

* This work was partly supported by the IT R&D program of MKE/KEIT [Development of the Core Technologies of General Purpose OS for Reducing 30% of Energy Consumption in IT Equipments] and research program 2012 of Kookmin University in Korea.

** Corresponding author

and extended attributes. In section 3, we introduce the design approach and implementation of YAFFS. Section 4 presents how to support extended attributes in YAFFS. Finally we conclude in section 5.

2 Linux Virtual File System Switch (VFS)

Android is built on top of the Linux kernel which provides basic and core system facilities such as security, memory management, process management, network stack, device drivers, file system management and etc. Among them, we focus on the file system management which plays a role as the intermediate between the kernel and real media. To implement extended attributes for YAFFS, we have to develop a set of functions which are used to initialize, set, get the extended attribute on the file system, and are utilized by the extended attribute functions of Linux VFS. This section introduces the role, data structures, implementation and extended attributes of Linux VFS. For more detailed description, we refer to read 3.

2.1 The role of VFS:

The VFS belongs to the kernel that handles all system calls related to a standard UNIX file system. Its main strength is to provide a common interface to several kinds of file systems.
For instance, assume that a user issues the shell command:

```
$ cp /floppy/test /tmp/test
```

Where */floppy* is the mount point of an MS-DOS diskette and */tmp* is a normal Ext2 directory. As shown in Figure 1, the VFS is an abstract layer between the application program and the file system implementation. Therefore, the *cp* program is not required to know the file system types of */floppy/test* and */tmp/test*. Instead, *cp* interacts with the VFS by means of generic system calls well known to anyone who has done UNIX programing.

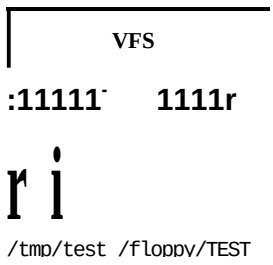


Figure 1 Copy file operation

The VFS supports three classes of file systems:

- Disk-based file systems: manage the memory space available in local disk partition.
- Network file systems: allow easy access to files included in file systems belongs to other networked computers.
- Virtual file systems: do not manage disk space, such as /proc file system, /dev/pts file system, and etc.

2.2 Data structures of VFS

Each VFS object is stored in a suitable data structure, which includes both the object attributes and a pointer to a table of object methods. The kernel may dynamically modify the methods of the object and hence it may install specialized behavior for the object. The VFS objects are 4:

- The superblock object: consists of *super* block data structure describing the abstract properties of the file system, such as its type (Ext2, Ext3, FAT32 and etc.), the physical device on which it resides, its total size, its mount point and a pointer to the root *dentry*.
- The dentry object: consists of *dentry* data structure carrying a file's name, a link to the dentry's parent, the list of subdirectory and siblings, hard link information, mount information, a link to relevant superblock object, and locking structure. It also contains a reference to its corresponding *Mode* object, and a reference count that reflects the number of process currently using it.
- The mode object: consists of *Mode* data structure containing information specific to a file, whether it is a regular file, directory of device. It also includes a link to the relevant superblock object, file permission, file type, file size, operations for use on mode by the VFS, device-specific information, information about how the file is memory-mapped, and etc.

2.3 **The implemetation of VFS**

The public interface of the Linux 2.6.x VFS consists of the header file *fs.h*, *namei.h* and *dcache.h* residing in *include/linux*. The implementation of system calls can be found in the *fs* subdirectory of the kernel source tree.
For the sake of brevity, we cannot discuss the implementation of all VFS system calls. However, it is very useful to sketch out the implementation of a few system calls to show how VFS's data structure interact. Let's reconsider the example proposed at the section II.A: a user issues a shell command that copies an MS-DOS file */floppy/test* to an Ext2 file */tmp/test*. The command shell invokes an external program like *cp*, which is assumed to execute the code in Figure 2

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/taw/test", O_WRONLY O_CREAT    0 TRUNC, 0600):
do (
   len = read(inf, buf, 4096);
   write(outf, buf, len);
) while (len):
close(outf):
close(inf);
```

Figure 2 Executing code of *cp* program

The *open()* system call

This system call is serviced by the *sys_open0* function, which receives path name, access mode and permission bit mask (if the file must be created) parameters, and performs the following:

Invokes *getname0* to read the file name form process address space.

1. Invokes *get_unused_fd0* to find an empty slot in *current->files->fd* and stores the corresponding index in the *fd* variable
2. Invokes the *filp_open0* function to create a new file object
3. Sets *current->files->fd[fd]* to the address of the file object
4. Returnf

The *read()* and *write()* system calls:

These two system calls are quite similar. Both require three parameters: a file descriptor *fd*, the address *buf* of memory area containing the data to be transferred, and a number *count* specifying how many bytes should be transferred. Of course, *read()* will transfer the data from the file into the buffer, while *write()* will do the opposite. Both system calls return the number of bytes that were successfully transferred or -1 to signal an error condition.

In detail, these system calls are serviced respectively by *sys_read0* and *sys_write0* functions which perform almost the same steps:

1. Invokes *fget0* to derive from *fd* the address of corresponding file object and increase the usage counter file->f *count*
2. Check whether the flags *in file->f mode* allow the requested access
3. Invoke *lock verift_area()* to check whether there are mandatory locks for the file portion to be accessed
4. If executing a write operation, acquire the *i_sem* semaphore included in the inode object
5. Invokes either *file->f op->read()* or *file->f op->write()* to transfer the data. Both functions return the number of bytes that were actually transferred.
6. Invokes thefi9uto to decrease to usage counter file->f *count*
7. Returns the number of byte successfully transferred

The close() system call

This system call receives a file descriptor *fd* parameter and is serviced by *sys close°* function which perform the following operations:

1. Gets the file object address stored in *current->files->fd[fdJ* (if it is *NULL,* return an error code)
2. Set *current->files->fdgcli* to *NULL* and releases the file descriptor *fd*
3. Invokes *filp _close()* to release any mandatory lock on the file and the file object
4. Return error code if there is any error, otherwise 0

2.4 Extended attributes (xattr) of an inode

Many new operating system features (such as access control lists, mandatory access controls, POSIX Capabilities, and hierarchical storage management) require file systems to be able associate a small amount of custom metadata with files or directories. Extended attributes have been introduced so as to support these features. These attributes are stored on a disk block allocated outside of any Mode. Each extended attribute has a name and a value. Both of them are encoded as character arrays with variable length.
Figure 3shows the layout of an Ext 2 file system block containing the extended attributes. Each attribute is split in two parts: the *ext2 xattr _entry* descriptor and the name of the attribute are placed at the beginning of the block, while the value of the attribute is placed at the end of the block. The entries at the beginning of the block are ordered according to the attribute names, while the positions of the values are fixed, because they are determined by the allocation order of the attributes.

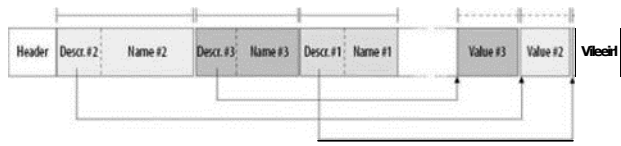


Figure 3 Layout of Ext2 file system block containing xattr

There are many system calls used to set, retrieve, list, and remove the extended attributes of a file. The *setxattrO, lsetxattrO,* and *fsetxattr0* system calls set an extended attribute of a file; essentially, they differ in how symbolic links are handled, and in how the file is specified (either passing a pathname or a file descriptor). Similarly, the *getxattrO, lgetxattrO,* and *fgetxattr0* system calls return the value of an extended attribute. The *listxattrO, llistxattrO,* and *flistxattr0* list all extended attributes of a file. Finally, the *removexattrO, lremovexattro,* and *fremovexattr0* system calls remove an extended attribute from a file.

3 YAFFS

In this section, we introduce the YAFFS, its design approach and implementation. For more detail, refer to 5 and 6.

YAFFS stands for "Yet Another Flash File System," a term coined by Charles Manning in 2001 when suggesting that the already clustered flash file system space could do with yet another offering — this time a flash file system designed from the group up to work with NAND flash.
YAFFS is designed to work in multiple environments which drive the need for portability. So that the primary strategies to improve portability include:

- No operating-system-specific features used in the maim code body
- No compiler-specific feature used in the main code body
- Abstract types and functions used to allow Unicode or ASCII operation

Simplicity is another key goal because it improves the robustness and the ease of integration and development. Primary strategies to achieve simplicity are:

- Single threaded model.
- Log structure makes for a simpler garbage collector and allocation methodology
- Abstractions that build layered code which is easier to understand and debug

The implementation of YAFFS is divided into four parts:

1. The file system algorithm
2. The interface layer to Linux VFS
3. The NAND interface which is a wrapper layer between the file system algorithm and the NAND memory access function
4. The portability functions which are wrapper functions for services such as memory allocation, etc.

4 Implementation and Experimental Results 4.1

Implementation

As mentioned in section III, the implementation of YAFFS is divided into 4 parts. To implement the xattr for YAFFS, we focus on the second part (the interface to Linux VFS) and its *inode_operations* structure which are shown in Table 1
Among these operations, there are only 4 functions — *yaffs_create, yaffs_mknode, yaffs_symlink* and *yaffs_mkdir* —that create new Mode objects. Moreover, the *yaffs create* and *yaffs_mkdir* functions recall *yaffs mknod* function. Therefore, we only have to modify two functions *(yaffs mknod* and *yaffs_symlink)* so that they initialize the xattr for an Mode when created.
For reusable purpose, we create a novel function called *yaffs_init xattr* which mainly calls security_inode_init_security and yaffs_set_xattrib functions. Finally, we insert it to the suitable place *ofyaffsimknod* and *yes' symlink.*

Table 1 (node operations

Operation Name	Description
yaffs_create(dir, dentry, mode, nameidata)	Creates a new disk inode for a regular file associated with a dentry object in some directory .11. Searches a
lookup(dir, dentry, nameidata)	directory for an inode corresponding to the filename included in a dentry object
yaffs_link(old_dentry, dir, new_dentry)	Creates a new hard link that refers to the file specified by old_dentry in the directory dir; the new hard link has the name specified by new_dentry.
yaffs_unlink(dir, dentry)	Removes the hard link of the file specified by a dentry object from a directory.
yaffs_symlink(dir, dentry, symname)	Creates a new inode for a symbolic link associated with a dentry object in some directory.
yaffs_mkdir(dir, dentry, mode)	Creates a new inode for a directory associated with a dentry object in some directory.
yaffs_rmdir(dir, dentry)	Removes from a directory the subdirectory whose name is included in a dentry object. .
yaffs_mknod(dir, dentry, mode, rdev)	Creates a new disk inode for a special file associated with a dentry object in some directory. The mode and rdev parameters specify, respectively, the file type and the device's major and minor numbers.
yaffs_rename(old_dir, old_dentry, new_dir, new_dentry)	Moves the file identified by old_entry from the old_dir directory to the new_dir one. The new filename is included in the dentry object that new_dentry points to.
yaffs readlink(dentry, buffer, buflen)	Copies into a User Mode memory area specified by buffer the file pathname corresponding to the symbolic link specified by the dentry.
yaffs follow link(inode, nameidata)	Translates a symbolic link specified by an inode object; if the symbolic link is a relative pathname, the lookup operation starts from the directory specified in the second parameter.
yaffs_put link(dentry, nameidata)	Releases all temporary data structures allocated by the follow_link method to translate a symbolic link.
yaffs truncate(inode)	Modifies the size of the file associated with an inode. Before invoking this method, it is necessary to set the i size field of the inode object to the required new size.
yaffs_permission(inode, mask, nameidata)	Checks whether the specified access mode is allowed for the file associated with inode.
yaffs setattr(dentry, iattr)	Notifies a "change event" after touching the inode attributes.
yaffs getattr(mnt, dentry, kstat)	Used by some filesystems to read Mode attributes.

yaffs_setxattr(dentry, name, value, size, flags)	Sets an "extended attribute" of an inode (extended attributes are stored on disk blocks outside of any inode).
yaffs_getxattr(dentry, name, buffer, size)	Gets an extended attribute of an inode.
yaffs_listxattr(dentry, buffer, size)	Gets the whole list of extended attribute names.
removexattr(dentry, name)	Removes an extended attribute of an inode.

4.2 Experimental results:

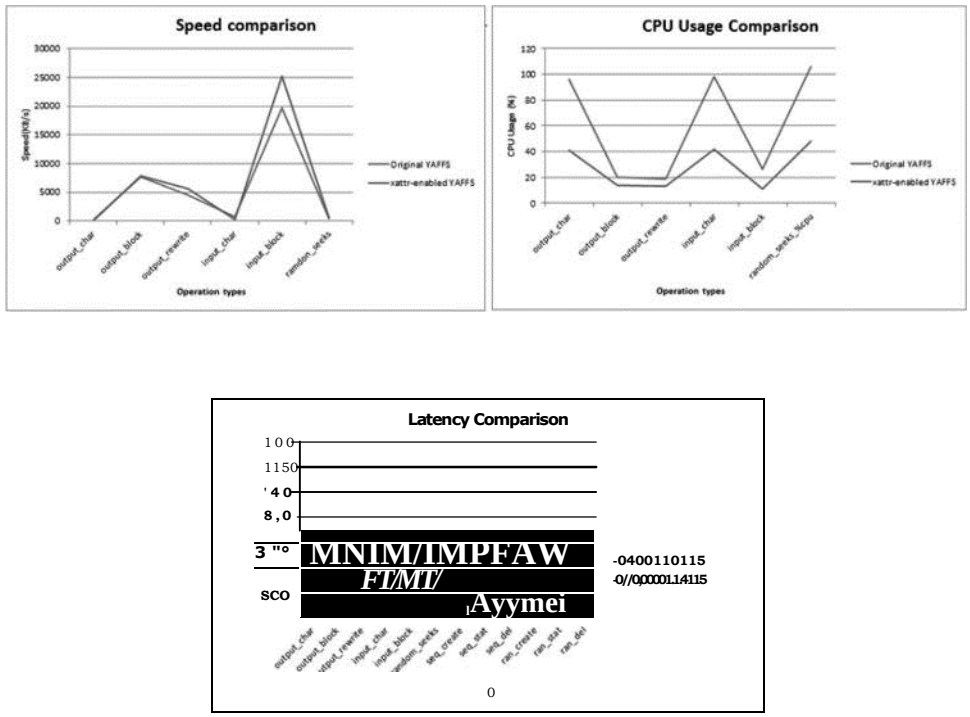
We ran a benchmark tool to check the performance overhead when using this new YAFFS on Android. We collected all benchmarking measurements on a Nexus S device, which based on 1GHz Samsung Exynos CPU, 512 MB RAM, 16 GB internal storage. The kernel is based on Linux 3.0 and Android platform is Ice Cream Sandwich 4.0.4

For evaluation, we ported and ran bonnie++ 7 in the device. Bonnie++ is a file system and hard disk performance benchmarking application for POSIX-compatible system. We execute each benchmark 10 times on the SMACA-enabled kernel 8 and 10 times on ordinary kernel. Table 2 lists the average benchmarking results in detail and Figure 4a, b and c graphically depict the comparison results of speed, CPU usage and latency respectively.

Table 2 Bonnie++ result

	Original YAFFS	xattr-enabled YAFFS
output char(KB/s)	251.9	104.7
output_char_%cpu	96.2	40.9
output_block(KB/s)	7678.6	7814.4
output_block_%cpu	20.3	13.6
output rewrite(KB/s)	4521.2	5700.3
output rewrite %cpu	19	13.3
input_char(KB/s)	705.7	279.5
input char_%cpu	98.3	41.9
input_block(KB/s)	19732.1	'5170.
input block_%cpu	26.3	**11**
ramdon_seeks(KB/s)	375.79	579.07
random_seeks_%cpu	105.5	48
output_char_latency(ms)	216.6	357.7
output block latency(ms)	878.9	747.9
output_rewrite_latency(ms)	783.5	765.1
input_char_latency(us)	81952.6	66646.5
input_block_latency(ms)	259.3	182.875
random seeks_latency(ms)	385.9	725.1
seq_create latency(ms)	865.6	991

seq_stat_latency(us)	2049.7	22895.5
seq_del_latency(ms)	402.2	890
ran_create_latency(ms)	486.3	1145.4
ran_statiatency(us)	203.3	16267.5
ran_del_latency(ms)	534.2	1038.1



a. b.

C.

Figure 4 Bonnie++ result

Although, the speed comparisons prove insignificant, there is significance in CPU usage and latency comparisons. The CPU usage comparison shows that the novel YAFFS requires less CPU usage than original one when reading, writing and seeking. However its latencies are longer than the original ones.

5 Conclusion and Future Work

It has been predicted that the next generation and breakthrough operating systems won't be on desktops or mainframes but on the small mobile device that we carry every day. The openness of these new environments will lead to new application and markets so as to enable greater integration with existing online service 9. However the increasing importance of data stored in mobile devices inevitably invites the more frequent security hole attacks. Android is a comprehensive open source operating

system for mobile devices. Its openness does not only give boom to its application market but also the vulnerability attack intents 8.

In this paper, we examined the root file system of Android called YAFFS and implemented the extended attribute for it. This support is really necessary because the extended attributes are required by many new operating system features (such as access control list, mandatory access control, and etc.) Currently, we have applied this novel YAFFS to our SMACA solution 8 and done some experiments. The results are shown in previous section. In the future, we will apply it for others solution such as SELinux, Tomoy, Smack, and etc.

References

1. Android security, Google Mobile Blog. http://googlemobile.blogspot.com/2012/02/android-and-security.html
2. Badger L, Stene D F, Sherman D L, and Walker K M. "A Domain and Type Enforcement UNIX prototype." In proceedings of the 5th conference on USENIX UNIX Security Symposium , pp 47-83, 1996.
3. Daniel P. Bovet and Marco Cesati. "Understanding the Linux Kernel 1st Edition." O'Reilly, 2001.
4. Andy Galloway, Grald Luttgen, Jan Tobias Muhlberg and Radu I. Siminiceanu. "Model-Checking the Linux Virtual File System." In Verification, Model Checking, and Abstract Interpretation, vol 5403, pp 74-88, 2009.
5. YAFFS I A Flash File System for Embedded Use. http://www.yaffs.net/
6. Charles Manning. "How YAFFS works". http://www.dubeiko.com/development/FileSystems/YAFFS/HowYaffsWorks.pdf
7. Bonnie++. http://www.coker.com.au/bonnie++/
8. Anh-Duy Vu, Jea-Il Han, Young Man Kim. "The Simple Mandatory Access Control for Android." In proceedings of International Converence on Computer, Networks, Systems, and Industrial Applications, to appear, 2012.
9. William Enck, Machigar Ongtang, Partrick McDaniel. "On lightweight mobile phone application cetification." In proceedings of the 16th ACM conference on Computer and communication security, pp. 235-245, 2009.