The Simple Mandatory Access Control for Android*

Anh-Duy Vu, Jae-I1 Han, Young Man Kim**

{anhduyvu, jhan, ymkim}@kookmin.ac.kr

Abstract -- The support of advanced mobile features (such as powerful processors, high memory and etc.) has paved the way for development of more attractive and convenient applications on smart phones, such as mobile banking and mobile online shopping applications. These applications require a highly secure environment which is basically provided by operating system and its own access control features. An access control scheme prevents system from activities that could lead to breaches of security. Although Android, a Linux-based and open-source mobile operating system hosted by Google, already provides various access control mechanisms those are inadequate for providing strong enforcement for system services which are executed on behalf of super user and could be tampered to devour the whole system. In this paper, we propose a new *Mandatory Access Control* (MAC) suitable for Google Android operating system called Simple MAC for Android (SMACA). SMACA is developed as a *Linux Security Module* (LSM) at kernel space and is absolutely transparent to Android user space so that there is no requirement to adjust current existing applications.

Key words: Android operating system, Android security, Mandatory Access Control (MAC), Domain Type Enforcement (DTE), Linux Security Module (LSM), Android malwares

1 Introduction

The support of advanced mobile features (such as powerful processors, high memory and etc.) has paved the way for development of more attractive and convenient applications on smart phones, such as mobile banking and mobile online shopping applications. These applications utilize some sensitive, private personal data and information of users such as bank certificates, credit card numbers, and etc. The use of such critical information requires a strong protection from any virus and malware.

Today, smart phone malwares are deceptive enough to spool itself as a legal mobile application. Their front-end services are so attractive that mobile users are deceived to download them without knowing their illegitimate background threat [1]. Therefore security protection in smartphone has become more and more essential and urgent like that of desktop systems. In last few years, a number of malwares have been designed to target and attack the mobile devices [2] [3].

Android is a Linux-based and comprehensive open-source operating system destined for mobile devices such as smart phones, tablets, music players, set-top-boxes, and etc. After the first instruction in 2007, it 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 scanner, Bouncer [4], for Android Market, it hardly ensures that users are completely free from all types of attack of malicious applications.

Android security framework incorporates several security mechanisms which could be clustered into two general groups: Linux security mechanisms and Android security policy enforcement mechanisms. However this framework does not enforce any security policy upon some system services, which are executed on behalf of super user, for example installd, adbd, void services. These privileged services potentially contain vulnerabilities so that there could be a situation that all private and personal resources stored on your smart phone are exposed.

In this paper, we extend Android security framework with a novel MAC model realizing a compact Domain and Type Enforcement (DTE) [5]. It extends the security enforcement upon the system services so it can mitigate malwares and system vulnerabilities. In the design and evaluation of the proposed module, this paper makes the following contributions:

*This work was partly supported by the IT R&D program of MKE/KEIT [10041753, 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

- We do an assessment to Android security framework and identify some limitations regarding its access control mechanism.
- We retrofit Android security framework with a novel MAC scheme called Simple MAC for Android (SMACA).

The remainder of this paper is organized as follows. Section 2 presents the overview of Android, its layered architecture, security framework and the limitations. In section 3, we describe the design of SMACA. Section 4 presents the mathematical validation. Then we describe the implementation of SMACA with benchmark in section 5. Finally we conclude in section 6.

2 Android overview

2.1 Android layered architecture

Android is an open source software stack for a wide range of devices including mobile phones, tablets, set-top-boxes and etc. It comprises of four separate layers: Linux kernel, core libraries, application framework, and application layers. Figure 1 depicts the major components of Android.

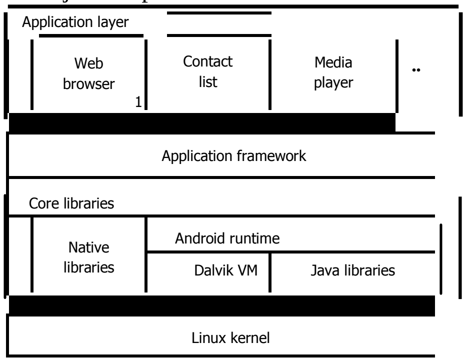

Figure 1 Android architecture

At the bottom of the stack is the Linux kernel which provides basic and core system facilities such as security, memory management, process management, network stack, device drivers, file system and etc. This layer is considered as an abstraction layer between the hardware and the rest of the software stack.
On top of the Linux kernel is the core libraries layer which consists of native libraries and the Android runtime environment:

- The native libraries consist of a set of C/C++ libraries used by various components of the Android system. These libraries provide certain core functionalities as graphic and signal processing, database, etc. Those are exposed to developers through the Android application framework.
- The Android runtime is composed of Java libraries and Dalvik virtual machine which is a Java virtual machine and tailored for the specific requirements of Android. Each instance of the virtual machine is used for an Android application execution.

At the next level up the stack is the Android framework. Here is the location of code compiled for and running on Dalvik virtual machines which provide services to multiple applications. Some entities running in this layer are (1) the Package Manager which is responsible for managing applications on the devices, (2) the Activity Manager which loads and manages Activity stack, and etc. This layer provides both third party and vendor developers full access to all framework APIs so all developers are free to take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and much, much more.
Android is shipped with a set of core applications including an email client, SMS program, calendar, etc. These applications are located in application layer which is at the top of the software stack. Third party applications are also installed in this layer. An Android application is developed in Java or C/C++ languages and is composition of activities [6], services [7], content providers [8] and broadcast receivers [9].

2.2 Android security framework

Android security framework incorporates several security mechanisms and could be classified into two general clusters: Linux security mechanisms and Android security policy enforcement mechanisms. The detailed features of these mechanisms are discussed more in later subsections

Linux kernel security mechanisms

The foundation of Android is the Linux kernel which has been broadly used for years by many corporations and security professionals in millions of security-sensitive environments. The Linux kernel security mechanisms provide Android with two key features including:

Applications sandbox

Android takes the advantage of the Linux DAC model as a means of identifying and isolating application resources. The Android system assigns a unique user ID (UID) to each application and runs it on behalf of that user in a separate process. In this way, the Linux kernel creates a sandbox and prevents applications from interfering with each other. This approach makes Android different from other traditional operating systems, where multi applications could run on behalf of one user and have the same permissions.

File system isolation

Each file in Linux environment is associated with the owner's UID, group ID and 3 tuples of *Read, Write,* and *eXecute (rwx)* permissions. The Linux kernel enforces these permissions and imposes the first tuple on the owner, whereas the second affects users that belong to the owners' group, and the third affects the rest of the users. The standard way that Android lays out the file system on a device is to create an application specific directory under the path */data/data/.* This directory is configured such that the associated application's UID is the owner and only the owner permissions are set; no other UIDs have access to it. Within this directory is files created by application when installed and executed.

Android security enforcement mechanisms

Reference monitor is a core component of Android security enforcement mechanisms, which is responsible to regulate the Inter Component Communication (ICC) [10]. A component unrestrictedly interacts with another one which belongs to the same application or another application that possesses the same UID. In the below, the first subsection will show you how to permit components of separated applications (do not share the same UID) to communicate with each other. The second one describes how to constrain whether a component can be accessible outside its application. Multiple applications can share the same ID if and only if they must specify the *sharedUserlD* attribute and are signed by the same digital signature which is called *Application signing mechanism* whose purpose will be discussed in the last subsection.

Application permissions

Similar to MAC models, Android enforces restrictions on specific operations that an application can perform via *permissions* (same as labels in MAC models) which are string representations. It has roughly 100 built-in permissions that control operations ranging from dialing the phone (CALL_PHONE), taking picture (CAMERA), using internet (INTERNET), and etc. Any Android application can declare additional permission. To obtain a permission, an application must explicitly request it. For example, if your application wants to communicate over the network, it would have an entry like Figure 2 in its AndroidManifest.xml file:

```
<?xml version."2.e" encoding="utf-8"?>
<manifest xmlns:android.¯http://schemos.android.com/opk/res/android¯
        package=¯com.wordpress.toptrinhnhung.IoneyBoole>


      <uses-permission androidmame="ondroid.permission.INTERNET" I>

     . . .
</manifest>
```

Figure 2 Request using internet permission

At installation, the system grants permission that the installed application requests based on checks of that application's signature against those of the applications declaring the permission and on the user's approval. After the user has installed the application and it receives its permission, it can no longer request any additional permission. An operation that hasn't received permission at installation time will fail at runtime

Component encapsulation
An Android application can encapsulate its components within the application content. This prevents other applications from accessing them, assuming that they bear a different UID. This occurs primarily through the definition of component's *exported* property. If the *exported* property is set to *false,* the component can be accessed only by the application that owns it. If it is set to *true*, external entities can access it. The entities are controlled through permissions as described in the previous subsection.

Application signing
Each application in Android is a package in an apk archive for installation. The Android system requires that all installed applications should be signed digitally. The signed apk is valid as long as its certificate is valid and the enclosed public key successfully verifies the signature. Signing applications in Android verifies that two or more applications are from the same author. The *shareUserid* mechanism and permission mechanism use this method to verify Signature and SignatureOrSystem protection-level permission.

2.3 Android security limitations:

We observe two limitation of the current Android security framework regarding to its design and security policy, respectively;

- Android locates security components at user space. It is impossible to protect system from attacks targeting at Android framework vulnerabilities. We need another security framework that secures not only applications but Android components as well.
- Android does not enforce security policy upon system services which are run on behalf of super user (root). It means that these privilege services can bypass all security check to access every resource on the device. In the case that an attacker can compromise one of them, the device is completely exposed to the attacker.

Android daemons running as root includes *initd, mountd, debuggerd, zygote,* and *installd [11]*. The *installd* daemon is in charge of unpacking and processing apk (Android installation package) files. We assume that an attacker detects vulnerability in this service, and he creates a malformed apk that can exploit a buffer overflow in the unpacking procedures. When this happens, the attacker can execute arbitrary code on behalf of installd's privileges (root privileges), and he can make phone calls, send message, copy your data to his server or even brick your device.
We propose an additional feature to Android security framework to remove this insufficiency. The proposed solution is an MAC implementation that enforces security policy upon not only user but system applications and daemons and is developed as Linux Security Module (LSM) located in kernel space.

3 Simple Mandatory Access Control for Android

In this section, we propose Simple Mandatory Access Control for Android (SMACA) to fix Android security flaws. SMACA is designed to meet the following specific design goals:

- SMACA should be kernel-based access control which can protect both applications and Android components.
- SMACA should automatically update the security policy when an application is installed or removed. Therefore, it does not require user to take care of security administration works.
- SMACA should not require any change to preexisting applications.

In order to satisfy the above requirements, we design SMACA as a compact implementation of DTE model and prove its capabilities by introducing several definitions presented in subsection *3.1* and its access control rules which are described in subsection *3.2.*

3.1 Methodology

All Android subjects (such as processes) and objects (such as files) are assigned an additional information, called security context or label, to each resource (both subject and object). Like SELinux [12], SMACA allows a subject to access (read, write, and execute) an object if and only if SMACA's policy allows the subject label to perform the operation on the object label. Unlike SELinux, the proposed module does not distinguish labels of subjects from those of objects. The labels are assigned according to the following rules:

1. *Kernel initialization:* the processes started at kernel initialization are labeled as *KERNEL_INIT.*
2. *System common resources: PUBLIC READ_WRITE* label is assigned to system resources that are required by all applications such as device drivers (located in */dev/).*
3. *System tools: PUBLIC EXECUTE* label is assigned to system executable objects (as those located in */system/bin.)*
4. *System development resources:* System libraries, header files and etc. are labeled as *PUBLIC READ.*
5. *Android permissions:* Resources attached with Android permissions (for example, internet, phone call) utilize the Android permissions name as their own labels.
6. *Android system components:* Android system processes and their resources have their own private labels. For example, label of *installd* daemon is *INSTALLD,* label of *initd* daemon is *INITD,* label of *vold* daemon is VOLD, and etc.
7. *Third party labels:* The UID of an application is assigned as the label for its subjects and objects. When a new application is installed, the system assigns it an UID and SMACA utilize this UID as the application's subjects and objects label

We define a mapping function that assigns a label to each subject and object.
Definition 1: Permission decision about each object access by a subject is made based on their labels. We define y as a function mapping a resource *r* (subject/object) to its label *1,* an element of universal set of labels *L.*

y(r)

where:
1 EL
L = {PUBLIC READ, PUBLIC EXECUTE, PUBLIC READ_WRITE ...}

Definition 2: SMACA specifies three basic access types including Read, Write and eXecute (r, w and x). We define a permissible access rule *R* presenting that a subject with label *S* is allowed with access of type *A* to object with label *0.*

R: (S, 0, A)

where:
S 00
S, 0 E L
A E {r, w, x)
A particular case that the label of the subject (or object) is set to *NULL* implies that a subject with any label is allowed with access type *A* to object with label *0* (or a subject with label *S* is allowed with access type *A* to an object with any label.)
Definition 3: We define a permissible access table *AT* that contains all permissible rules R_x.

AT = {R!, R2, R.3.-}

Definition 4: To determine the permission of an access trial, we define *6* as a granting function that answers Y (granted) if and only if there is an access rule R_t in the access table *AT* allowing it; otherwise *N* (rejected)

6(S, 0, A) 9 Y v N

where:
S: subject label
0: object label
A E {r, w, x}

3.2 SMACA access control rules

The following rules are the enforced SMACA rules, in order:

1. *Kernel initialization:* Any access requested by subject labeled *KERNEL_INIT* is always permitted. As mentioned above, this label is only assigned to processes started at kernel initialization and they are conventionally considered as trusted processes. Therefore SMACA allows them to interact with all resources.

   R_1: (KERNEL INIT, NULL, {r, w, x})

2. *Access to system development objects:* Any read access requested on objects labeled *PUBLIC READ* is permitted. These objects consist of system libraries, header files and etc. which are required by all system and third party applications so that SMACA must allow all subjects to read them.

   R_2: (NULL, PUBLIC READ, r)

3. *Utilization of system tools:* As mentioned above, the system tools called as *toolbox* and located in */system/bin* are labeled *PUBLIC EXECUTE.* SMACA allows all subjects to execute them.

   R_3: (NULL, PUBLIC EXECUTE, x)

4. *Access to system common resources:* SMACA grants all read-and-write accesses requested on system public objects such as drivers... which are labeled *PUBLIC READ WRITE.*

   R_4: (NULL, PUBLIC READ WRITE, {r, w})

5. *Internal access.*• Any access requested on an object by a subject with the same label is always permitted.

   R_5: (1, 1, {r, w, x})

   where:
   1 E L

6. *Defined access:* Any access requested that is explicitly defined in the access table *AT is* always granted.

   R_a: $(1_1, 1_2, A)$

   where:
   lb 1_2 EL
   $1_1$012
   A e {r, w, x}

7. Otherwise, any request is rejected

The first five rules are called built-in rule (or default rules) and never modified. The sixth one represents rules added to security policy to make a third party application work correctly.

The format of a security policy rule is:

Subject Object Access

where:
Subject: the label of the subject,
Object: the label of object
Access: r, w or x denoting Read, Write and eXecute operations respectively.

SMACA security policy describes explicitly how a subject interacts with an object. And there is no implicit rule available. For example:

app_14	*app_15*	*rw*
app_15	*app_16*	*rw*

These rules tell that an application with label *app_14* can observe and modify objects with label *app_15* and similarly an application with label *app_15* can observe and modify objects with label *app_16.* But they do not imply that the application with label *app_14* can observe and modify object with label *app_16.*

4 Mathematical Validation

In the first subsection, we examine an Android vulnerability of *vold* daemon which was first exploited by Sebastian Krahmer [13] and is also reported as CVE-2011-1823 [14]. And in the second subjection, by mathematical equation, we prove that SMACA can prevent such vulnerability. This vulnerability is chosen to validate the effectiveness of our proposed SMACA because this attack is really powerful such that more and more Trojan and malware are created to exploit this security hole as reported in [15], [16] and [17].

4.1 Overview of void vulnerability

void daemon is a system service (run on behalf of root) that is in charge of managing disk volumes. It receives disk mounting/un-mounting command via Netlink messages from kernel. The problem is that it does not verify message content and sender before using it.
In [18], the author clearly denotes that the *void* daemon receives a parameter called *PARTN*, converts it into integer value and uses the result as index of *mPartMinors* array without checking who sent this value as well as whether the conversion result is non-negative.
The attack, which was already developed as a utility called Gingerbreak, consists of following steps:

1. Collecting information necessary for exploitation.
   The purpose of this step is to:

- Identify process **ID** (PID) of *void* process. To do this, attacker has to examine */proc/net/netlink* to get a list of PIDs of processes communicating with kernel via Netlink socket. After getting the list, the attacker read each */proc/<PID>/cmdline* file to find out which *PID* represents the *void* process.
- Identify addresses and values of interest. The necessary addresses and values are the

  — .GOT address range of *void* daemon which can be obtained by examining */system/bin/void* file
  – The *system* address which is the shell command can be taken by examining */system/lib/libc.so* file.
  – The index value of *mPartMinors* array which can be done successfully by restarting *logcat* service, sending malicious data (negative index values) to *void* daemon, and finally examining the *logcat* file.

2. Sending carefully crafted Netlink message to *void.* In this step, the attacker

- Creates a *setuid-root shell* which allows the attacker to update his privilege as those of super user.
- Bruteforces the .GOT range by continuously sending a Netlink message that passes the *system* address as the *MINOR* value and the negative index to *PARTN*

3. Updating *shell commands* as root. After the second step, the *void* daemon will execute the *setuid-root shell.* So the attacker that already has the permission to remount */data* directory makes *shell commands* as root binary to get root privilege.

4.2 SMACA vs. vold vulnerability

In this subsection, we will represent the Gingerbreak hacking tool as mathematic equations along with the proof showing that SMACA can absolutely protect our device from Gingerbreak and the others which utilize similar techniques. In this validation, we accept a prerequisite assumption that:

- Gingerbreak has passed all security checks of Android framework. In other words we assume that SMACA is the last gate that Gingerbreak has to bypass to steal the root privilege.
- SMACA utilizes the default permission access table *($AT=\{R_1, R2, R3, R_4, R_S\}$)*
- *GGB* is the label of Gingerbreak
- *LOG _CAT* is the label of *logcat* process

1. Reading */proc/net/netlink* file: In SMACA, this access request is evaluated as follows:

$$c5(GGB, root, r) - N$$

   SMACA absolutely rejects this request because of two reasons:

- The corresponding labels of the subject and the object are different. The label of Gingerbreak is GGB and the label of */proc/net/nedink is root*
- The permissible access table *AT* does not contain any rule that allows the label *GGB* to read the *root.*

Therefore this step fails, the attack process is stopped here and our system is securely protected. However, let's assume that the attacker finds some advanced techniques to bypasses this protection and continues his process. Let's proceed to the second sub-step.

2. Reading */proc/<PID>/cmdline* files. These files are created by running applications of third parties or root, and they have different labels. To do this scanning job successfully, Gingerbreak requires the below requests to receive permission, those are turned out to be rejected.

   (5(GGB, root, r) 4 N
   o(GGB, app_1, r) 4N

   The default security rules of SMACA will reject all of them except the request accessing to Gingerbreak resource. This means that Gingerbreak cannot detect the PID of *void* daemon so the attack is defeated here. Once again, we assume that the attacker uses some advanced techniques to overcome this protection for further analysis. Let's keep going to the third sub-step.

3. This sub-step accesses three files: */system/bin/vold*, */system/lib/libc.so,* and *logcat* files. Among these requests, the access to */system/lib/libc.so* is permissible because (1) this file is labeled as *PUBLIC READ* and (2) the first SMACA access rule allows this access.

   o(GGB, PUBLIC READ, r) 4 Y

   However the access to */system/bin/void* file is prohibited. The */system/bin/vold* file is labeled as *PUBLIC EXECUTE* so SMACA can only allow any execute access request on this file:

   o(GGB, PUBLIC EXECUTE, r) N

   And, the read access request to *logcat* file of Gingerbreak is rejected by SMACA default policy:

   o(GGB, LOG_CAT, r) 4 N

In conclusion, the Gingerbreak completely fails to do collecting-information job and thus cannot do any further exploitation to our system.

5 Implementation and benchmark 5.1

Implementation

SMACA is implemented as a LSM [19] and integrated into Linux kernel. The process of handling a request is denoted in Figure 3. This approach offers some benefits:

- Reducing number of codes that must be added and modified in the Linux kernel
- Avoiding "Time Of Check, Time Of Use" (TOCTOU) errors

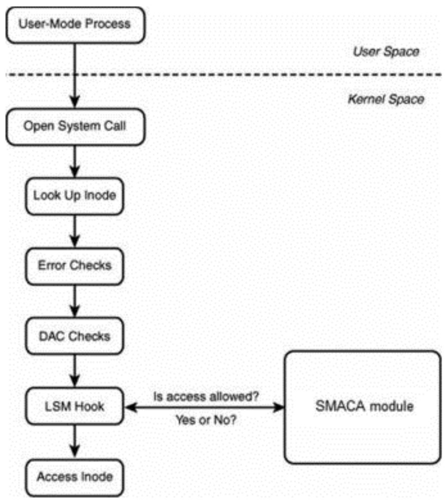

Figure 3 SMACA modules as LSM

The SMACA module consists of an implementation of LSM hooks that manages resource access requests, a pseudo file system and a set of utility functions. At early initialization of the system, the pseudo file system is mounted, read the SMACA policy and loads it to the kernel. The Figure 4 show a part of *init.rc* file which denotes how to mount *smaca* file system.

```
on early-init
    * Sec init and its forked children's oom_adj. write
    /proc/./oom adj -16

    start ueventd

* create rnuntpoints
    mkdir /smaca
    mount smacafs smacafs /smaca

    mkdir /mnt          root system


on init
```

Figure 4 init.rc file

For optimization, a process locally caches its permitted rules describing that the process can do certain operations (read, write and execute) on some object labels. Concretely, when receiving a request to access a resource from a process:

1. SMACA consults the process' permitted rules which are cached in its *security field.* If SMACA cannot find a rule that permits the request and the cache version is different form version of SMACA kernel policy, go to step 2. Otherwise the request is rejected.
2. If SMACA cannot find a rule that allows the request in the kernel policy and the kernel policy version is different from version of SMACA pseudo file system policy, go to step 3. Otherwise it will update the process' cache and version and allow the request.
3. Reload the pseudo file system to kernel, update the version and redo step 2.

Figure 5 depicts this algorithm.

SMACA uses extended attribute (xattr) to store object labels. However the traditional root file system of Android (YAFFS) does not support this extension. Therefore we implemented this attribute for YAFFS. The detail of this implementation is described in [20].

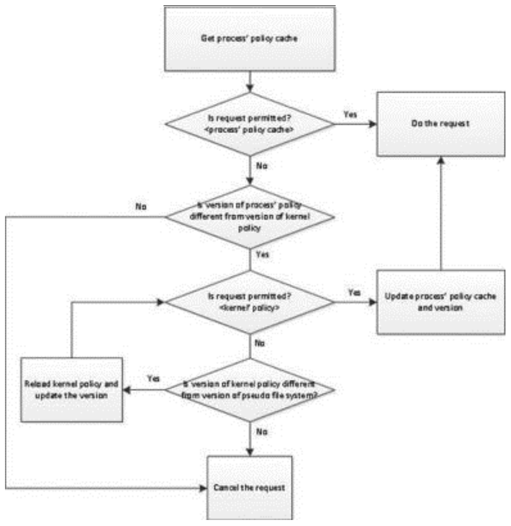
Get process' policy cache
Is request permitted? <process' policy cache>
Yes
Do the request
No
Is version of process' policy different from version of kernel policy
No
Yes
Is request permitted? <kernel' policy>
Yes
Update process' policy cache and version
No
Is version of kernel policy different from version of pseudo file system?
Yes
Reload kernel policy and update the version
No
Cancel the request

Figure 5 Request handling procedure

5.2 Benchmark

We ran several benchmarks to check the performance overhead when using SMACA 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 utilized lmbench tool [21] and suite which is a set of micro-benchmarks for low-level Linux functionalities. We execute each benchmark 10 times on the SMACA-enabled kernel and 10 times on ordinary kernel. 0 lists the average benchmarking results in detail and Figure 6 graphically depicts the latency results.

We observed notable slowdowns in operations requiring additional permission check such as open, close, read, and state. The remaining actions showed moderate slowdowns.

Table 1 Evaluation results

Latency	Without SMACA	With SMACA
Simple syscall (us)	0.34579	0.34634
Simple syscall (us)	0.58543	2.12526
Simple write (us)	0.56917	0.734141M1
Simple stat (us)	6.37074	10.01306
Simple fstat (us)	0.84493	1.21608
Simple open/close (us)	8.57084	14.25067
Simple handler installation (us)	0.80304	0.80195
Simple handler overhead us	2.8999	6.79976
Protection fault (us)	0.5986	1.31667

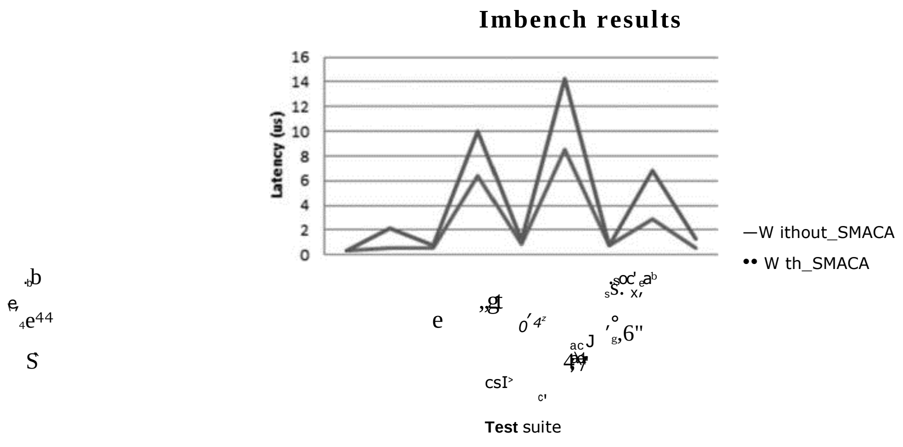

Figure 6 lmbench results

6 Conclusion

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 [2]. 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.
In this paper, we identified the limitations of Android security platform and proposed a novel security feature called SMACA to remove them. The SMACA (1) provides an implementation of DTE model of MAC that can protect both applications and Android framework, but (2) does not require user to play security administration role and (3) does not require any change to preexisting applications.
In addition, we have proved and demonstrated the SMACA protecting abilities against Gingerbreak via mathematic equation and evaluate the performance of SMACA version 1.0. In the future, we will optimize the code to reduce the latency slowdown.

References

1. Hammad Banuri, Masoom Alam, Shahryar Khan, Jawad Manzoor, Bahar Ali, Yasar Khan, Mohsin Yassee, Mir Nauman Tahir, Tamleek Ali Quratulain Alam, Xinwen Zhang. "An Android Runtime Security Policy Enforcement Framework". In Personal and Ubiquitous Computing, pp. 1-11, 2011.
2. 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.
3. Jazilah Jamaluddin, Nikoletta Zotou, Reuben Edwards. "Mobile phone vulnerabilities: a new generation of malware." In 2004, IEEE international symposium on consumer electronics, pp. 199-202, 2004.
4. Android security, Google Mobile Blog. http://googlemobile.blogspot.com/2012/02/android-and-security.html
5. 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.
6. Android activities component, Android developer site. http://developer.android.com/guide/topics/fundamentals/activities.html
7. Android services component, Android developer site. http://developer.android.com/guide/topics/fundamentals/services.html
8. Android content providers component, Android devloper site. http://developer.android.com/guide/topics/providers/content-providers.html
9. Android application fundamentals. http://developer.android.com/guide/topics/fundamentals.html
10. Machigar Ongtang, Stephen McLaughlin, William Enck, Patrick McDaniel. "Semantically rich application-centric security in Android." In proceedings of the 2009 Annual Computer Security Applications Conference, pp. 340-349, 2009.

11. Asaf Shabtai, Yaval Fledel, Yuval Elovici. "Securing Android-powered mobile devices using SELinux." In Security & Privacy, Vol 8, Issue 3, IEEE, pp.35-44, 2010.
12. Security-Enhanced Linux. http://www.nsa.gov/research/selinux/
13. Yummy yummy GingerBreak! http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html
14. CVE-2011-1823 at National Vulnerability Database. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823
15. New Android malware variant can remotely root phone. http://threatpost.com/en_us/blogs/new-android-malware-variant-can-remotely-root-phone-040412
16. Security Alert: New variants of Legacy Native (LeNa) identified. http://blog.mylookout.com/blog/2012/04/03/security-alert-new-variants-of-legacy-native-lena-identified/
17. Angry birds space trojan uses Gingerbreak. http://rootzwild.com/news/ Jarticles/angry-birds-space-trojan-uses-gingerbreak-r635
18. Android void mParMinors[] signedness issue. http://xorl.wordpress.com/2011/04/28/android-vold-mpartminors-signedness-issue
19. Chris Wright, Crispin Cowan, Stephen Smalley, James Morris and Greg Kroak-Hartman. "Linux Security Module: General Security support for the Linux kernel." In proceedings of the 11th USENIX Security Symposim, pp. 17-31,2002.
20. Anh-Duy Vu, Jae-I1 Han, Young-Man Kim. "A Security Enhancement for Android's Root File System". In proceedings of International Converence on Computer, Networks, Systems, and Industrial Applications, to appear, 2012.
21. lmbench tool. http://www.bit