Journal of Engineering and Applied Sciences 13 (Special Issue 2): 3059-3066, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

Key Resources Integrity Verification Scheme on Android Platform

[1]Yongzhen Li, [1]Zhenzhen Wang and [2]Hyung-Jin Mun
[1]Department of Computer Science, Yanbian University, 133002 Yanji, China
[2]Department of Information and Communication Engineering, Sungkyul University,
14097 Anyang-City, Republic of Korea

Abstract: Nowadays with Android mobile phones occupied a large market share, the security of application programs has gotten more and more attention. To protect the security and integrity of application programs on android platform, a flexible key resource integrity verification scheme was proposed. The scheme principally utilized security-file which is created by the security-file generator on the developer's server to verify resources integrity. Then APPs downloaded the security-file from the server and the procedure of integrity verification is processed by Java Native Interface (JNI). The scheme also provides error correction module by downloading updates in the case of a validation failure. The experimental result showed that this scheme had good ability against decompiling attack and tampering attack and the performance of the application has not been affected.

Key words: Integrity verification, security-file, decompiled, Java native interface Android, application, affected

INTRODUCTION

With the popularity of mobile internet Android smartphone is developing rapidly. The Android operating system has been supported by the major mobile phone manufacturers and the vast number of developers because of its Linux Kernel and open source. In our daily life, smartphones have become more and more significant as one of the communication tools.

Under such a background an application program on Android platform has gotten explosive growth but its security problem has been intensified. However, the ubiquity of smartphones loaded with the android system makes them more attractive targets to attackers. Currently, differentiated forms of malwares exist in a lot of smartphone platforms, also in Android (Younan *et al*., 2006). According to McAfee Labs statistics in the fourth quarter of 2014, mobile malware has broken through 600 million which increased by 14% than the third quarter (Huang *et al*., 2013). The unsafe applications not only infringe on fruits of labor of developers but also pose a serious threat to user privacy, even seriously affecting the future development of the Android application market. Recently, the safety of Android application has become an issue to be addressed in researching not only domestically but also, globally (Shu *et al*., 2014). But the study of Android application protection is still in its infancy.

Application protection technology has caused widely public concern in the early of the 1980's. Kent (1980) puts forward the concept of application protection in the perspective of developers. He pointed out that piracy prevention, tamper-proof and reverse engineering prevention is the main purpose of application protection. However, there is no acknowledged technology for tamper-proof technology at present in terms of integrity verification, it is worth of studying further (Felt *et al*., 2011).

Background

Hash function: The hash function, H is the function that has an arbitrary length message, M and a fixed length output, H (M) to M is called "message digest". (Davies and Price, 1980). A good hash function has following characteristics, the outcome of the function toconsiderable inputs is equalized and calculated distinctively in random values. Figure 1 illustrates the common-operation of a cryptographic hash function.

Hash function for message authentication is proposed by Mitchell *et al*. (1989). Message authentication is a mechanism or a service which is used in verifying the integrity of a message. Message authentication checks the validity of the data sender and whether the received data is the message sent by the sender.

Security-file: Nowadays, hash function algorithm applies to the data integrity verification widely. Hash function is able to map a message in arbitrary length into the hash value in fixed length. This study provides the ability of

Corresponding Author: Hyung-Jin Mun, Department of Information and Communication Engineering, Sungkyul University, 14097 Anyang-City, Republic of Korea

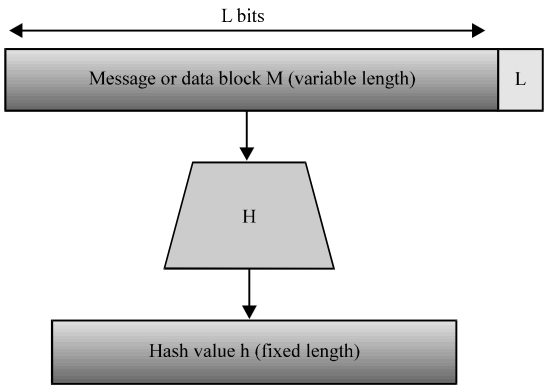

Fig. 1: Hash function (h = H(M))

Table 1: Format of security-file

Security-file (*.sec)	File names
Protected file 1	File name (fileName)
	Creation data (fileData)
	File size (fileSize)
	Hash value (hashValue)
Protected file 2	File name (fileName)
	File size (fileSize)
	Creation data (fileData)
	Hash value (hashValue)
Protected file n	File name (fileName)
	File size (fileSize)
	Creation data (fileDate)
	Hash value (hashValue)

the integrity verification using a security-file which is a collection of hash values including all key resources specified by the developer. The security-file's each entry stores resource name, the create date of file, size of the file and the hash value, it has a special file suffix (.sec) acquiescently. The format of the security-file as in Table 1.

In the whole integrity verification scheme, security-file plays a decisive role. To protect the security of security-file, the client download security-file from the server when starting for the first time, after that the client only verifies whether hash value is consistent with the corresponding resource name.

Android NDK and JNI: In Android, most of Android application developers usually prefer using Java with Android NDK to solely using the Java language (Kim *et al*., 2012). Using native libraries from C/C++ code in Android Apps. android NDK (Ratabouil, 2015) known to be a toolset is able to embed components. Although, to use native code rarely enhances the performance of application android NDK is efficient in terms of CPI-intensive operations, signal processing and physics simulation. It is efficient when re-using a large scale of legacy C/C++ code (Gordon, 1998).

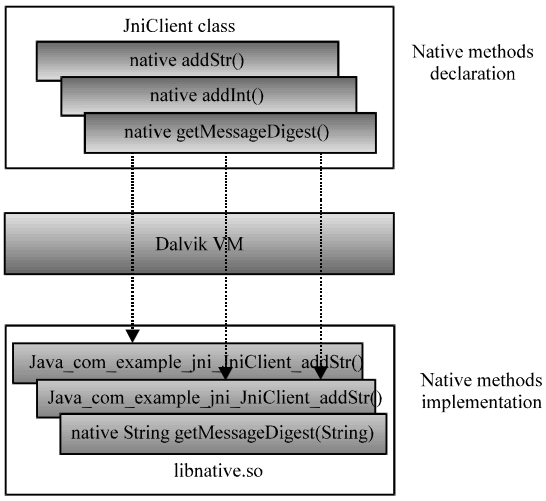

Fig. 2: Example native function calls via. JNI

JNI (Anonymous, 2017) is an abbreviation of Java Native Interface, it is a programming framework that makes Java code running in the Java Virtual Machine (JVM) call and be called by native libraries and application which are written in C/C++ and other assembly languages (Lee *et al*., 2011). Since, the Java language itself is not suitable for development of security software, Google has provided Android NDK (Android Native Development Kit) to help developers to access native C/C++ code from Java layer (Budd, 1999).

JNI is used to reuse an important large mass of native code written in C/C++ to enhance the application performance using time-critical code. There are negative effects resulted from JNI including loses of the portability and safety of Apps (Lee and Jeon, 2010). But there are more positive effects and following are advantages of using JNI technology.

The Java layer code can easily be decompiled while C/C++ code more difficult. Using C/C++ code to develop application has higher execution efficiency. Using C/C++ code to develop dynamic link libraries is easier to transplant.

Figure 2 illustrates one example of native function calls via. JNI. Figure 3 describes three different native method declarations from JniClient class with the Android application. They need to be declared in a class with respect to an Android application. To write a "native" directive in prior to writing the return type and name of the method is necessary. Figure 2 depicts three different native method implementations in libnative. So, shared library with native C code. In an Android application environment, native methods have to have one to one correspondence to the native method declaration in a class (Qian *et al*., 2016).

```
public class JniClient {
    static public native String addStr(String strA, String strB);
    static public native int addInt(int a, int b);
    static public native String getMessageDigest(String msg);
}
```

Fig. 3: JniClient class

By developing an Android application, we are able to utilize the native code, shared libraries, under the standard System. LoadLibrary() call. The argument of System.load Library() is the name of the library. We should pass in "native" if we wanted to use the shared library, "libnative.so".

MATERIALS AND METHODS

Proposed key resources integrity verification scheme: In Android, integrity protection did not consider other resources in the existing scheme such as images, audio and so on other than dex file and did not provide the right resources to download in the case of a validation failure. The scheme in this study, not only can specify a particular resource dynamically by developers but also provides error correction module by downloading updates.

Preparation-generating security-file: A security-file is generated dynamically according to developer's requirements. We developed a tool to help developers produce security-file, the specific process as shown in Fig. 4. The step to generate security-file is:

- Choosing a project directory path
- Choosing to be protected files or resources in "Project files" such as "logo.png" and then press the "Add" button, the selected items will be shown in "Added files"
- Generating the security-file

Communication between client and server: When the application starts, it will check whether there is security-file if not it will download security-file from the server. In order to prevent data from falsifying during the transmission, the specific process as shown in Fig. 5 and the whole validation process is as follows:

- The server calculates hash value of the security-file as the digital signature
- The server generates encrypted hash value by encrypting the hash value calculated in step j with a private key
- The security-file and encrypted hash value are sent to the client
- The client decrypts the encrypted hash value with public key which is saved in local
- The client calculates hash value of the security-file
- The client verifies two hash values obtained in step m and step n, respectively

If the result of step o is true, we can ensure that the security-file has not been falsified during transmission. Even though an attacker would tamper security-file, the private key cannot be deducedas far as the server is not exposed because tampered security-file is encrypted with the private key.

Integrity verification: Integrity verification was designed in Java native layer, using JNI to prevent the application from reverse engineering. The times of integrity verification are determined by the number of key resources, times add when the application running to key resource. The overall flow chart is shown in Fig. 6. Resources integrity verification is implemented using JNI, mainly requires the following steps:

Step 1: The Android application ensures whether there is security-file, if not then it will send a request to the server to download the security-file.

Step 2: There is a security-file and then the application needs to verify its integrity. If the result is false, the application also needs to send a request to download the unbroken security-file.

Step 3: The Android application loads program to run on the hypothesis that step 1 and 2 are true.

Step 4: Once the running program meets protected resource, the application should check the hash value of protected resource utilizing the security-file. If the validation fails, the application should download unbroken and corresponding protected resource from the server.

Step 5: Repeating step 3 and 4 until the application come to the end.

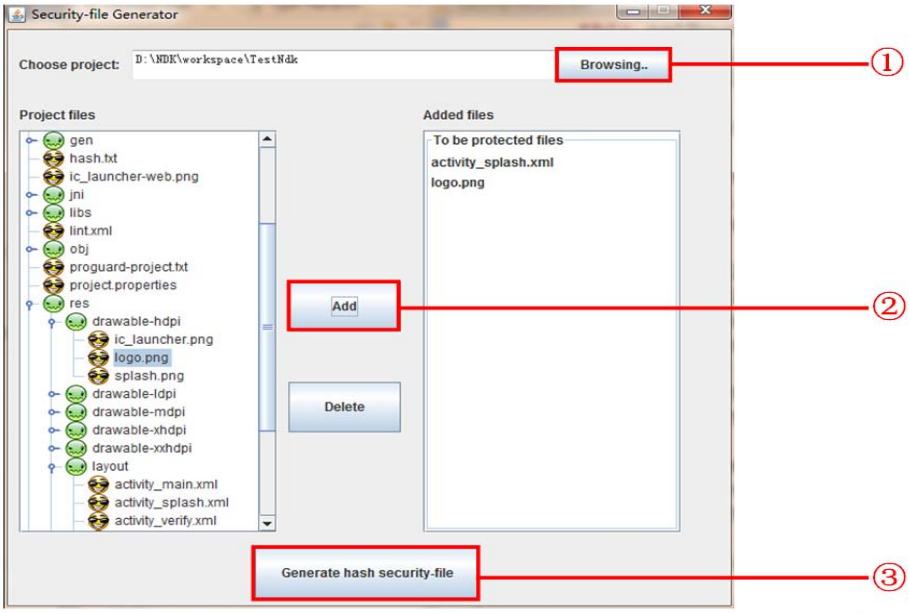

Fig. 4: Security-file generator

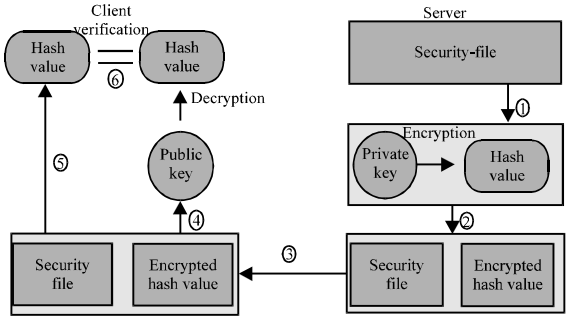

Fig. 5: Transmission process

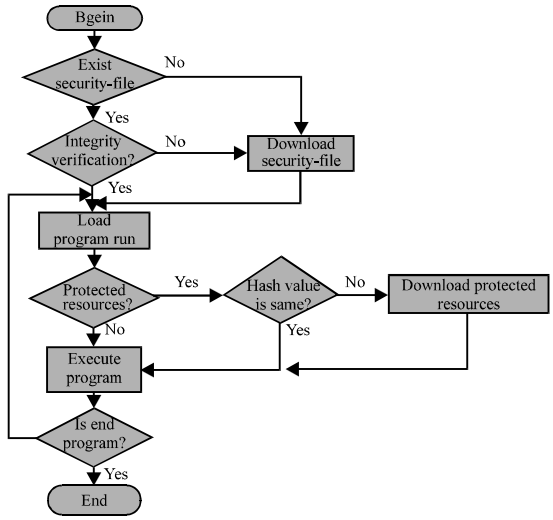

Fig. 6: The overall flow chart of scheme

RESULTS AND DISCUSSION

This study has conducted experiments from the aspects of feasibility and the defense of the compiler and has made a comparison about JNI invoking and Java invoking. The feasibility experiment will prove the scheme is practical and can ensure that the critical resources are not tampered. When security-file exists and hash value of key resources is correct, the application will run correctly and there is no difference between ordinary applications.

Decompiler experiment: Decompiler experiment proves that JNI invoking can enhance the ability against decompiling and make the application more secure. As we all know an apk file is a compression file containing various resources that are comprised of an Android application. Android code will be packaged into.dex file by apktool, dex is Android Dalvik executable program. We can decompile. Dex file with baksmali and smali tools, then we obtain some smali files. The smali files composed of Dalvik byte-code which shows instructions being used as well as registers, objects and literal values which secures readability. The smali files also contain messages about instance variables, data types, functions and so on.

What we need for attention is the high lighted part in red horizontal line and box in Fig. 7, we can see clearly that the hash algorithm is SHA 1 and the implementation of SHA 1 also can be found which is viewed by JD-GUI

```
.line 21
const-string v3, "hash.sec"

invoke-static {p0, v3}, Lcom/example/minatest/FileReadWrite;
    ->getContentFromAssets(Landroid/content/Context;Ljava/lang/String;) Ljava/lang/String;

move-result-object v0

.line 22
.local v0, "filebeanString":Ljava/lang/String;
new-instance v2, Lcom/example/hash/SHA1;

invoke-direct {v2}, Lcom/example/hash/SHA1; -><init>() V

.line 23
.local v2, "sha1":Lcom/example/hash/SHA1;
invoke-virtual {v2,v0}, Lcom/example/hash/SHA1; ->Digest(Ljava/lang/String;) Ljava/lang/String;
```

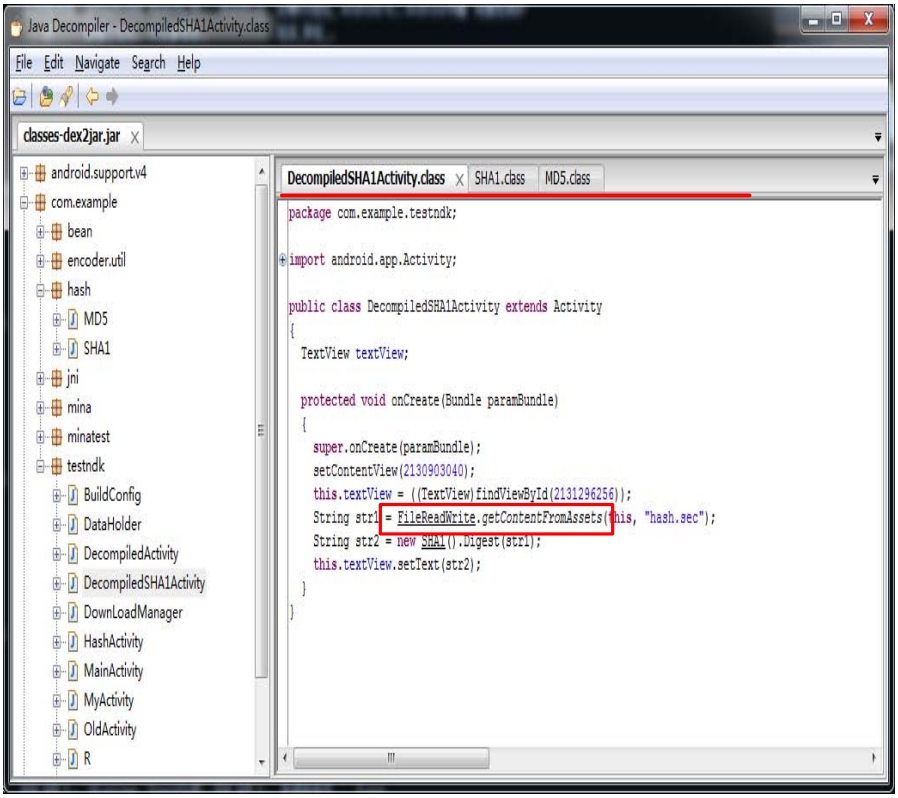

Fig. 7: Decompiled code and source code using Java

tool. However, if we use JNI invoking to verify integrity, although the code is decompiled, attackers do not known what hash algorithm used by developers, we can see it from Fig. 8. JniClient class only has method declarations that is public static native String get Message Digest (String msg). Because native method is not in Java layer, even if be decompiled also won't reveal native method.

Performance experiment: We present the different aspects regarding performance between Android application with JNI and Android application using the same algorithm written with Java language solely. We estimate JNI communication lag incurred from JNI. The application of ours employs the shared library not performing computing operation but passing a string to the App. JNI invoking spends 38 msec in 105 times, so, we

```
.line 26
const-string v2, "hash.sec"

invoke-static {p0, v2}, Lcom/example/minatest/FileReadWrite;
    ->getContentFromAssets(Landroid/content/Context;Ljava/lang/String;) Ljava/lang/String;

move-result-object v0

.line 27
.local v0, "filebeanString":Ljava/lang/String;
invoke-static {v0}, Lcom/example/jni/JniClient;
    ->getMessageDigest(Ljava/lang/String;) Ljava/lang/String;
```

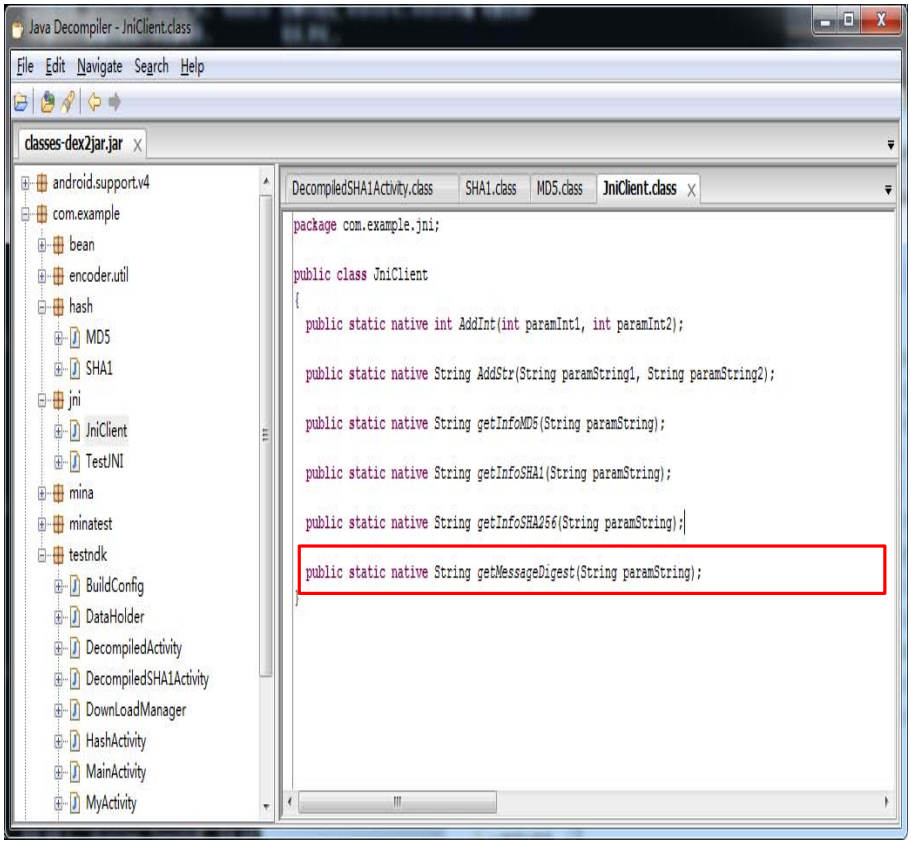

Fig. 8: Decompiled code and source code using JNI invoking

can assume that once time consumption is minimal, the slight delay can be ignored to some extent. The experimental data about the efficiency of JNI invoking and Java invoking is elaborated in Table 2. To improve the credibility of the experiment, we did two experiments using two different hash algorithms, the one is MD5 and the other is SHA1. The histogram (Fig. 9) is drawn by

Table 2: The experimental data about efficiency of JNI and Java

	MD5		SHA1	
Variables	JNI (msec)	Java (msec)	JNI (msec)	Java (msec)
500	77.470	142.13	79.60	123.56
700	104.75	194.08	113.44	163.34
1000	147.81	285.12	155.64	235.89
2000	309.41	635.96	324.59	567.08

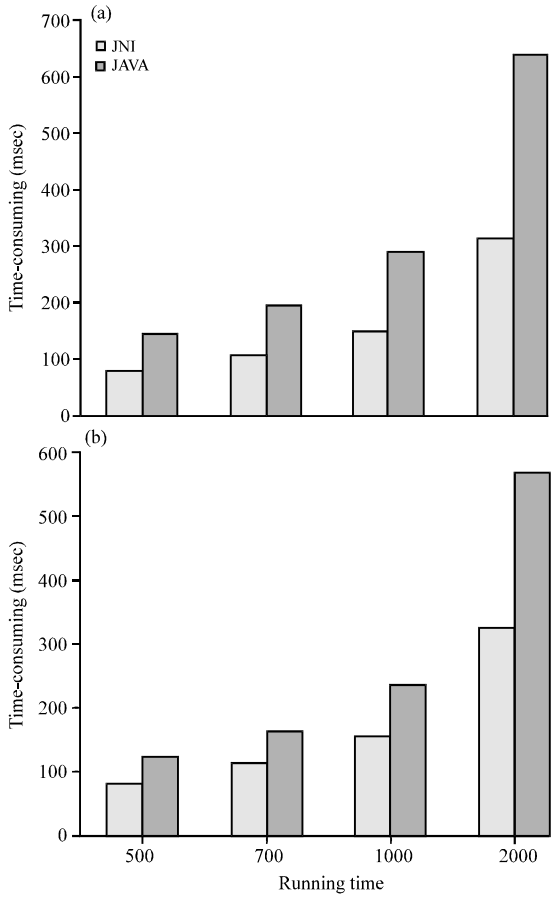

Fig. 9: Comparison diagram about efficiency of JNI and Java

MATLAB based on the experimental data in Table 2 to display clearly. According to the comparison diagram and after analysis and comparison, the efficiency of JNI invoking of MD5 is 1.918 times higher than Java invoking and SHA1 is 1.564 times.

CONCLUSION

Key resources integrity protection scheme is crucial in protecting application resources. It not only prevents key resources from tampering and also provides error correction function. By providing error "correction" function, it requires the application must be connected to the server, it is the prerequisite for the protection scheme can be executed correctly. The experimental results showed that the critical resource integrity protection scheme can provide more secure on the Android platform, at the same time through JNI invoking can effectively prevent the application from decompiled.

REFERENCES

Anonymous, 2017. Java native interface. Wikimedia Foundation, Inc., San Francisco, California, USA.https://en.wikipedia.org/wiki/Java_Native_Interface.

Budd, T., 1999. C++ for Java Programmers. Pearson Education, London, England, UK., ISBN:978-81-317-6472-5, Pages: 284.

Davies, D.W. and W.L. Price, 1980. The Application of Digital Signatures based on Public key Cryptosystems. National Physical Laboratory, Teddington, England, UK.

Felt, A.P., M. Finifter, E. Chin, S. Hanna and D. Wagner, 2011. A survey of mobile malware in the wild. Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, October 17, 2011, ACM, Chicago, Illinois, USA., ISBN:978-1-4503-1000-0, pp: 3-14.

Gordon, R., 1998. Essential JNI: Java Native Interface. Prentice-Hall, Upper Saddle River, New Jersey, USA., ISBN:9780136798958, Pages: 499.

Huang, H., S. Zhu, P. Liu and D. Wu, 2013. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In: Trust and Trustworthy Computing, Huth, M., N. Asokan, S. Capkun, I. Flechais and L. Coles-Kemp (Eds.). Springer, Berlin, Germany, ISBN:978-3-642-38907-8, pp: 169-186.

Kent, S.T., 1980. Protecting externally supplied software in small computers. Ph.D Thesis, Massachusetts Institute of Technology Cambridge, Cambridge, Massachusetts.

Kim, Y.J., S.J. Cho, K.J. Kim, E.H. Hwang and S.H. Yoon *et al.*, 2012. Benchmarking Java application using JNI and native C application on Android. Proceedings of the 12th International Conference on Control, Automation and Systems (ICCAS'12), October 17-21, 2012, IEEE, JeJu Island, South Korea, ISBN:978-1-4673-2247-8, pp: 284-288.

Lee, S. and J.W. Jeon, 2010. Evaluating performance of Android platform using native C for embedded systems. Proceedings of the International Conference on Control Automation and Systems (ICCAS), October 27-30, 2010, IEEE, Gyeonggi-do, South Korea, ISBN:978-1-4244-7453-0, pp: 1160-1163.

Lee, Y.H., P. Chandrian and B. Li, 2011. Efficient Java native interface for Android based mobile devices. Proceedings of the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), November 16-18, 2011, IEEE, Changsha, China, ISBN:978-1-4577-2135-9, pp: 1202-1209.

Mitchell, C., D. Rush and M. Walker, 1989. A remark on hash functions for message authentication. Comput. Secur., 8: 55-58.

Qian, Q., J. Cai, M. Xie and R. Zhang, 2016. Malicious behavior analysis for android applications. Intl. J. Netw. Secur., 18: 182-192.

Ratabouil, S., 2015. Android NDK: Beginner's Guide. Packt Publishing Ltd., Mumbai, India.

Shu, J., J. Li, Y. Zhang and D. Gu, 2014. Android app protection via. interpretation obfuscation. Proceedings of the IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC'14), August 24-27, 2014, IEEE, Dalian, China, ISBN:978-1-4799-5079-9, pp: 63-68.

Younan, Y., D. Pozza, F. Piessens and W. Joosen, 2006. Extended protection against stack smashing attacks without performance loss. Proceedings of the 22nd Annual Conference on Computer Security Applications (ACSAC'06), December 11-15, 2006, IEEE, Miami Beach, Florida, pp: 429-438.