Smart Card Based Protection for Dalvik Bytecode — Dynamically Loadable Component of an Android APK

Muhammad Shoaib, Noor Yasin, and Abdul G. Abbassi

***Abstract*—The global smartphone market is growing at a brisk pace. Android, an open source platform of Google has become one of the most popular mobile operating systems. Android apps generate lot of revenue which is increasing every year. The reverse engineering of Android applications is much easier than owing to the use of open source platform. Therefore, it becomes important to protect applications running on Android from attackers. The goal is to minimize software flaws and use anti-reverse engineering techniques. In this paper, we present a protection scheme based on obfuscation, code modification and cryptographic protection that can effectively counter reverse engineering on the Android platform. Our approach aims at making it tough for a reverse engineer to get the business logic performed by an Android application.**

***Index Terms*—Software protection, Android, dalvik bytecode, reverse engineering, code obfuscation, anti-reverse engineering, cryptography.**

I. Introduction

The use of software applications has increased a lot in last few decades and they have become a necessity. From mobiles and computers to electronic devices, software applications are all around us. Owing to their wide use, software industry has become one of the largest and most important businesses that can generate huge revenue. The importance of software applications and their unprecedented growth together with the vulnerabilities found in them, make them a prime target of attackers resulting in attacks like reverse engineering, illegal use and distribution and tampering. Thus, software industry is facing the big threat of piracy. Attacks on software are done for variety of reasons like economic gains, for fun and even for satisfaction. Attackers bypass the registration/licensing process, reverse engineer binaries and files, tamper with them and redistribute the software. Business Software Alliance (BSA) reported that software piracy has resulted in a loss of 63.4 billion to software industry [1]. More alarming is the fact that this loss is on the rise and increasing every year as it surged from $58.8 billion in 2010 to $63.4 billion in 2011.

In recent years Android platform, which is developed by the "Android Open Source Project" has become one of the most popular systems for mobile devices and the market for Android applications has rapidly grown in variety and financial volume. This platform is designed in a way that developers can upload and publish an app on Android market without a review from Google and users can easily download and install new applications. The Android's smartphone mobile app revenues reached nearly $6.8 billion by the end of 2013, almost doubling its revenues from the previous year [2]. This results in an increasing demand to safeguard intellectual property of developers and protect Android applications files from piracy.

Majority of security solutions have been designed and deployed primarily focusing on the client side interests. Firewalls, Intrusion Detection & Prevention Systems (IDPS), antivirus, digitally signed software, etc are few examples of security applications that provide software protection at the user end but do not protect against the software vulnerabilities exploited by attackers and reverse engineers. Therefore we need solutions that can cater to developers' needs and protect their interests against attacks like reverse engineering, Break Once Run Everywhere (BORE), illegal tampering, and unauthorized use of software. Security organizations as well as researchers around the globe are working in three directions to achieve protection against these attacks. The first group is working on solutions based on cryptographic and obfuscation techniques. Second group is working on software licensing laws and implementing Digital Rights Management (DRM) to protect the intellectual property rights and the last group is working on making software secure by design.

Although different techniques and third party tools have been proposed in the past for the protection of Android apps, but to the best of our knowledge, no protection scheme is completely secure and cost effective. Further, if it is assumed that the attacker has enough time and resources, then any protection scheme can ultimately be broken. Therefore, the aim here is to make the process of reverse engineering hard if not impractical, so that it may introduce enough delay to sell out legitimate copies of the software and generate substantial revenue. In this regard, we present a software protection scheme for Android applications that utilizes the benefits of encryption and obfuscation. Our solution is capable of restricting the process of reverse engineering and illegal distribution for a reasonable amount of time and hence ensures that desired level of protection of Android applications is achieved.

The rest of this paper is organized as follows. In Section II we discuss the Android architecture, sandboxing and Android application build process. Section III covers current threats to Android apps, discusses a generic reverse engineering model and existing methods and tools used for Android app protection. Section IV introduces our protection method which is based on encryption and obfuscation. In Section V we analyze our work and conclude this paper.

Manuscript received August 10, 2014; revised February 2, 2015.

All the author are with School of Electrical Engineering and Computer Science (SEECS) NUST Campus H-12, Islamabad, Pakistan (e-mail: 11msccsmshoaib@seecs.edu.pk, 11msccsnyasin@seecs.edu.pk, Abdul.ghafoor@seecs.edu.pk).

II. Android Basics

A. Android Architecture

When an Android program is compiled, all of its parts are packaged into one file called Android Application Package file (APK). An APK is a zipped file formatted package with .apk extension and contains:

- Dalvik Executable file (.dex file containing dalvik bytecode)
- Resources & Assets
- Certificates
- Manifest file

The dalvik bytecode contains the program code for an Android application and is executed by the Dalvik Virtual Machine (DVM). Apart from executing dalvik bytecode, DVM also provides the ability to execute code which is not part of the dalvik bytecode i.e. by calling native functions within shared objects. The dalvik bytecode is comparable to java bytecode but is designed specifically for Android applications. Owing to limitations of resources on mobile devices, dalvik bytecode is more efficient and compact than java bytecode. Resources and assets of an APK file include bitmap files, sounds and other static data that is used by Android application. Every application has an AndroidManifest.xml file in its root directory which presents essential information about the application to the Android system. An APK file, in order to be installed on any device, has to be digitally signed with a certificate whose private key is held by the developer. Self Signed certificates can also be used without the need of a Certificate Authority. The signing process adds no security and merely identifies the application developer. It happens automatically when we use Eclipse with the Android Development Toolkit (ADT) plugin. The Android system comes with an optimizer and verifier tool called "dexopt" [3]. When an Android application is installed on any device, dexopt will optimize and verify the dalvik bytecode for efficient execution on underlying architecture. This process is called optimization and the resultant dex file is called "odex".

B. Sandboxing

When it comes to Android security, a key concept is "secure sandbox". The Android application Sandbox isolates app data and code execution from other apps. This is achieved by assigning a unique user ID (UID) to each app running as a process. By default no application has the permission to interfere with another app's resources and private data. Only processes with same UIDs can share resources. In order to allow an application to interfere with another application's sandbox, permissions must be explicitly declared up front before the app is installed and cannot be changed after installation.

The only way to break out of application sandbox is by compromising the linux kernel which is the operating system Android is based upon. This is called rooting the device where each app will have a root level access and can modify other apps data as well as the kernel. Rooting an Android device renders all the security mechanism null and void.

C. Android Application Build Process

A lot of work has been done in past to secure binaries and executables [4]-[6] but not much has been achieved for security of mobile applications especially Android apps (APK files). As Android apps are java based so it is quite easy to decompile and reverse engineer them by using few tools and having a basic knowledge. Application developers therefore are interested in protecting the business logic of their applications so that it is hard to understand what an application does and how its functionality is implemented. An overview of Android application build process is shown below:

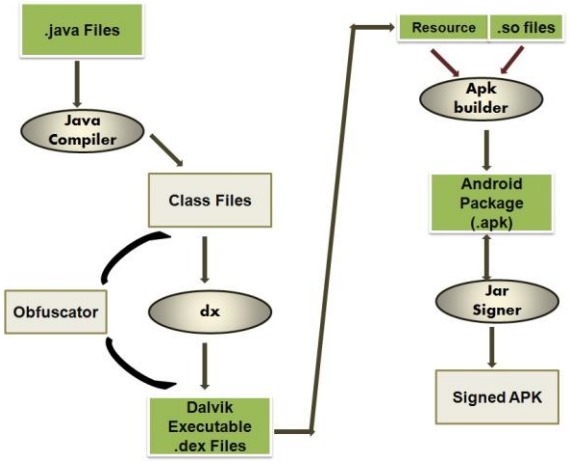

Fig. 1. Android application build process.

An Android application needs to go through many steps and requires different tools for building an APK file which is ready for deployment. The first step of the build process, as shown in Fig. 1 is compilation of ".java" files into ".class" files using Java compiler. Class files contain java bytecode and at this step obfuscation can be applied to this java bytecode. In the next step java bytecode is transformed into dalvik bytecode using a utility "dx" which comes with Android Software Development Kit (SDK). The application of "dx" utility on class files results into a single dex file called "classes.dex". It is possible to apply further obfuscation on dalvik bytecode at this stage using Proguard or any other obfuscation tool. The ApkBuilder then constructs an APK file from the "classes.dex" file and adds further resources like images and ".so" files. The ".so" files are shared objects which contain native functions that can be called from within the DVM. In the last step "jarsigner" adds developer's signature to the APK. The signed application can finally be deployed on an Android device.

III. Related Work

Some of major threats to Android apps are:

- Unauthorized API access
- Malwares and viruses
- Repackaging and selling
- Piracy
- Stealing app code and assets

The first threat to generate and modify source code is reverse engineering of software. When we speak of reverse engineering an Android application, we primarily mean to reverse engineer the dalvik bytecode located in the classes.dex file. There are many tools to interact and analyze

Android applications. Smali/baksmali assembler and disassembler are useful tools for reverse engineering of Android apps. Dex2Jar is a tool that is used to transform dalvik bytecode i.e. dex files to normal jar files. APKTool is a well known application for reverse engineering of Android applications. JD-GUI is a cross platform utility which can display java source code of ".class" files. JAD is a tool that is used to extract source code from class files. Androguard is python based tool which is used for reverse engineering of APK files. DexDump and Dexter are also useful code analysis tools which are quite effective for reverse engineering of Android apps. All these tools are discussed in the work of kim *et al*. [7]. IDAPro is another useful tool for effective decompilation of binaries [8]. The decompiling procedure of an APK using Dex2Jar and JD-GUI is shown in the Fig. 2 below:

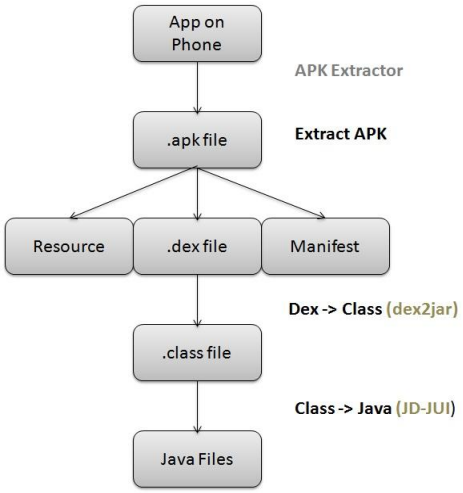

Fig. 2. Decompiling an APK.

Android app protection is relatively a new topic. Android applications can be protected in three ways:

- Anti piracy
- Anti tampering
- Anti reverse engineering.

The anti piracy technique for application protection includes techniques like licensing, Digital Rights management (DRM) and software watermarking. Google has introduced license verification library [9], a tool library that protects apps from being stolen by third parties. Operators like Amazon and Verizon have introduced their own DRM options to protect their apps from being pirated or copied. Software watermarking is also an effective tool for anti piracy [10].

Anti tampering in Android is mainly achieved via a signature mechanism which verifies integrity of the APK file. We can also verify the integrity of classes.dex file to make sure that source code has not been tampered with. Encryption can also be used to prevent tampering if key exchange and key storage is carefully dealt with.

In order to protect the software from malicious attacks, anti reverse engineering techniques are used for defense. Since Android apps are java based so protection techniques based on obfuscation that are used to prevent reverse engineering of java bytecode have been researched and applied to dalvik bytecode as well. Code obfuscation is a technique in which a program is automatically transformed in such a way that its functionality remains the same while it is more difficult to reverse engineer [11], [12]. Obfuscation is a useful and cost effective technique and it doesn't require any special execution environment. Moreover it is believed to be more effective on Android system [13], [14]. Patrick Schulz in his work "Code Protection in Android" [15] discusses some possible code obfuscation methods on the Android platform using identifier mangling, string obfuscation, dead code insertion, and self modifying code. Ghosh *et al*. [16] have discussed a code obfuscation technique on the Android platform that aims at increasing the complexity of the control flow of the application so that it becomes tough for a reverse engineer to get the business logic performed by an Android application. Kundu has also worked on some obfuscation techniques like clone methods, reordering expressions and loops, changing the arrays and loop transformations [17]. These techniques make it hard for the attacker to understand the logic behind decompiled code.

For protection of Android applications, obfuscation has also been used in different code protectors. There are a number of java obfuscators like proguard, dexguard, allaatori, etc that obfuscate the code by removing unused code and by renaming classes, fields, and methods to semantically obscure names. These obfuscators are used to create apps that are optimized, faster, more compact, and more difficult to crack [18].

We have seen that anti reversing techniques that have been implemented for defense of APKs are mostly based on obfuscation. The reason cryptographic protection is not generally used is that the existing execution environments do not support execution of encrypted files. Therefore the encrypted file has to be decrypted before execution and the attacker can intercept the code when it is decoded for execution into the internal memory. Therefore, we have to customize/tweak the existing execution environment to enable it to run the encrypted software. In this paper we have proposed a solution based on obfuscation, code modification and cryptographic techniques for software protection. The cryptographic protection prevents static analysis while obfuscation and code modification delay dynamic analysis. A combination of both can effectively make reverse engineering hard even for an experienced attacker.

IV. Proposed Solution

In the first step of our design, the application developer publishes the app on the application repository. On receiving a download request from the user, app repository redirects it to Identity Management Server (IDMS) which requires the user to register on the IDMS. The user then has to access the app repository to download the app containing encrypted dex file. The app can only be downloaded upon authentication from the IDMS. Once the file is downloaded it can be executed in a customized environment using a Dexfile loader. The encrypted file is decrypted on the fly using a key provided by the user via smartcard. An overview of design is given in Fig. 3.

A. Registration

The user first registers with the registration server (IDMS)

using registration web page [19]. This registration information is sent using SSL protocol to the Web Server. Once the registration process is complete, a group ID is generated for the user which acts as a passphrase for generating a Groupkey for dex file encryption/decryption later. This ID is same for a group of users with size n and is changed for every subsequent group. For example if we take a group size of 10 (i.e. n=10), then GroupID will remain same for 10 users and will change for next group of users. Thus generated key is same for a group that consists of n users.

In order to download the app, the user sends a download request to app repository. The app server verifies that user is registered at the IDMS and allows the download to proceed after authentication as shown in Fig. 4.

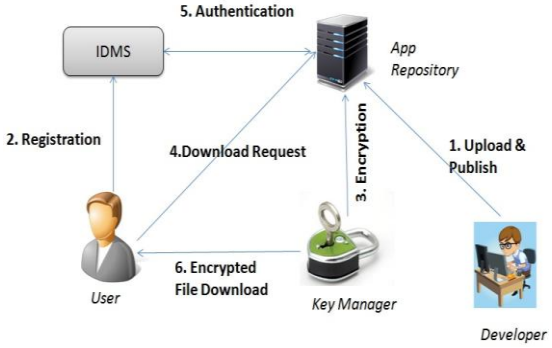

Fig. 3. Design overview.

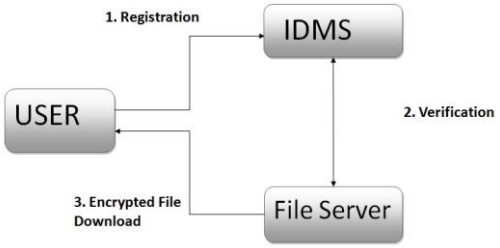

Fig. 4. Registration, Authentication & file download.

B. Encryption and Decryption

Our protection technique has two main parts:

- Packer (Encryptor)
- Unpacker (Decryptor Stub)

1) Packer

A perfect obfuscator would result in transformation of an application in such a way that it is impossible to analyze it and extract any information from it. But obviously it is not possible to generate such an ideal transformation. A close enough result can be achieved by using a technique often used by malware i.e. packing [20]. A packer takes an executable or binary file, encrypts it so it cannot be analyzed by the attacker/analyst unless decrypted. We know that the main code of an Android application is stored in a dex file and we can use encryption to protect the dex file. In Android, encrypted dex file can easily be generated using AES [21]. The GroupID which was generated at registration step is used as a passphrase for generation of key using Password Based Encryption (PBE) [22]. This key is used to get a .dex file with an encrypted dalvik bytecode. The application containing encrypted dex file is then stored on the server. The PBE generated Groupkey is stored by key manager using public keys of the registered users. Thus each user can use his private key to extract the GroupKey which will be used for decryption later. In this way the symmetric key can be distributed to multiple recipients of the group who have registered at IDMS and downloaded the application with encrypted dex file.

2) Unpacker

The unpacker or Decryptor stub is an important component of our design and it must be executed when starting an application. It performs some key functions like fetching the dex file, decrypting and loading it into the memory and executing it. The loading of dex file is generally achieved by using reflection which loads dex file from a certain location in a currently running process. The problem with this simple approach is that data has to be decrypted on the system prior to execution and this decrypted data can easily be copied by the analyst, rendering the whole protection scheme futile. The unpacker in our solution can be used to load encrypted dex file and decrypt it on the fly for execution.

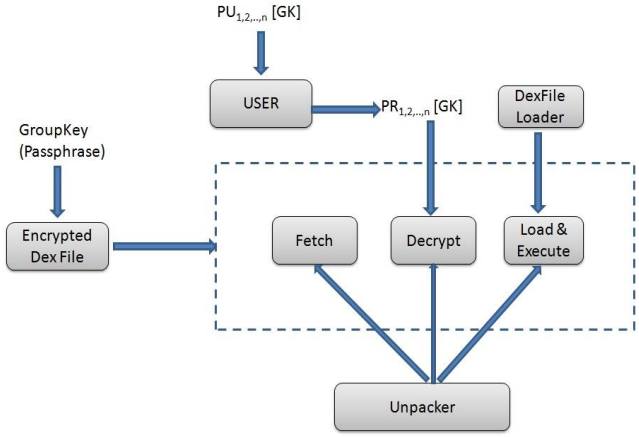

Fig. 5. File encryption & decryption process.

Our main aim is to make hard the analysis of unpacker and protect dex file containing dalvik bytecode. This can be achieved by obfuscating the bytecode and encrypting dex file using a cryptographic function like AES. Android does not allow loading an encrypted dex file. In order to overcome this problem, we first decrypt the encrypted dex file into a bytearray and then load from it. Loading a dex file from bytearray into the Dalvik Virtual Machine (DVM) is also not possible in Android by using standard procedures. Therefore we access the private method "***private static int openDexFile(byte[] fileContents)***" of DexFile to achieve this task [23]. Private methods in Java can be accessed either by calling ***setAccessible(true)*** on method before invoking it or using Java Native Interface (JNI). In this way we do not have to store the decrypted dex file on local storage rather it is present only in the volatile memory and for rest of the time it remains encrypted. The symmetric key is also stored and entered via a smartcard by the user and is not stored in the application which makes protection scheme quite secure. The file encryption and decryption process is shown in the Fig. 5.

So far the only place where the bytecode is present unencrypted is in the process memory. An expert attacker can root the device and trace process memory to get useful information. We know that the DVM as well as the dalvik bytecode present in the process memory cannot be modified

at run time but we can circumvent that restriction by using native code that can be called from within the DVM using JNI to modify the dalvik bytecode. Thus, if the attacker roots the device and accesses process memory still it won't be easy to analyze the intercepted code owing to obfuscation and code modification.

V. Conclusion

The protection technique proposed in this paper is for the applications where we have to deliver the app code and resources to the users in order to run the application in offline mode. By applying the approach proposed in this paper we are making it difficult for the attackers to reverse engineer and access the bytecode. Once we modify the execution environment, the dex file is decrypted on the fly into a bytearray from which it is loaded and then executed. The advantage of this approach is that code is not decrypted into the internal memory and is only present unencrypted in the volatile memory. Another advantage of our solution is that the decrypted dex file has not to be stored in optimized form and thus can be deleted to prevent unauthorized access. The proposed technique also provides some degree of protection in case the device is rooted and process memory is accessed. Obfuscation and the use of JNI for code modification together make analysis hard. Apart from dalvik bytecode the attacker also has to analyze the native code which further delays the reverse engineering process. An important technique that we have used in this research is that decryption key is provided via smartcard by the user. Thus we do not have to store the key with the application. The decryption key is secured by encrypting it with users' public key so only authorized users can get access to the key. Cryptographic protection using Groupkey ensures that an attacker has to be a registered user which may also be used in future for tracking the attacker.

The protection scheme presented in this paper has some limitations as well. It cannot be used at large scale and is ideal only for a company or small group of people. Our proposed scheme does not protect the resources and assets of an Android app and protects only the dex file containing dalvik bytecode. The size of dex file should not be very large for this scheme to be applicable. This technique requires the Android device to be compatible with smartcards.

Software protection is a challenge and protection of Android apps is still a relatively new research. Android is basically built on an open source platform owing to which reverse engineering an Android app is easy as compared to applications based on some other platform. Therefore, when dealing with highly secure systems and information, it is better to put all the business logic and code on the server side. Using a real time service or remote server to deliver content is the best practice to prevent source code from reversing.

References

[1] Business software alliance. (October 28, 2013). [Online]. Available: http://globalstudy.bsa.org/2011

[2] Abi research. (September 18, 2013). [Online]. Available: https://www.abiresearch.com/press/android-mobile-app-revenues-will-reach-68-billion- Last

[3] DexOpt. [Online]. Available: https://android.googlesource.com/platform/build/+/donut-release/tools/dexpreopt/dexopt-wrapper/

[4] S. Kent, "Protecting externally supplied software in small computers," Ph.D. thesis, M.I.T., September 1980.

[5] Ultraprotect 1.05. [Online]. Available: http://www.brothersoft.com/ultraprotect-18090.html/

[6] Exestealth protector. [Online]. Available: http://www.webtoolmaster.com/exestealth.htm/

[7] K. Yekyung, "Framework for analysis of android malware," Diss. University of Akron, 2014.

[8] IDAPro. [Online]. Available: http://hex-rays.com/idapro

[9] License verification library. (September 18, 2013). [Online]. Available: http://developer.android.com/google/play/licensing/index.html Last

[10] W. Zhou, X. Zhang, and X. Jiang, "AppInk: Watermarking android apps for repackaging deterrence," in *Proc. 8th ACM SIGSAC symposium on Information, Computer and Communications Security*, Hangzhou, China, May 2013, pp. 1-12.

[11] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, no. 7, pp. 64-71, July 2003.

[12] P. Sivadasan, P. SojanLa, and N. Sivadasan, "Jdatatrans for array obfuscation in java source codes to defeat reverse engineering from decompiled codes," in *Proc. 2nd Bangalore Annual Compute Conference*, Bangalore, India, January 2009, p. 13.

[13] A. Venkatesan, "Code obfuscation and virus detection," M.S. Project, Dept. Comp. Science, San Jose State University, California, USA, 2008.

[14] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," *Information Hiding*, pp. 270-284, 2011.

[15] P. Schulz, *Code Protection in Android*, Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany, 2012.

[16] S. S. Ghosh, S. S. R. Tandan, and K. Lahre, "Shielding android application against reverse engineering," *International Journal of Engineering Research and Technology*, vol. 2, no. 6, June 2013.

[17] D. Kundu, "JShield: A java anti-reversing tool," PhD thesis, San Jose State University, California, USA, 2011.

[18] Y.-X. Piao, J.-H. Jung, and J.-H. Yi, "Server-based code obfuscation scheme for APK tamper detection," *Security and Communication Networks*, 2014.

[19] A. G. Abbasi and S. Muftic, "Cryptonet: Integrated secure workstation," *International Journal of Advanced Science and Technology*, vol. 12, pp. 1-10, November 2009.

[20] Malware obfuscation using code packing. [Online]. Available: http://www.foocodechu.com/?q=node/55/

[21] Cipher. [Online]. Available: http://developer.android.com/reference/javax/crypto/Cipher.html/ Last Accessed: 2013-02-11.

[22] Nelenkov blog spot. [Online]. Available: http://nelenkov.blogspot.com/2012/04/using-password-based-encryption-on.html

[23] Dexfile. [Online]. Available: https://android.googlesource.com/platform/libcore-snapshot/+/ics-mr1/dalvik/src/main/java/dalvik/system/DexFile.java/

Muhammad Shoaib is from Dera Ismail Khan, a district of KPK province of Pakistan and has a distinguished academic career. He received his B.S. degree in computer system engineering from Ghulam Ishaq Khan Institute of Engineering Sciences & Techlology (GIKI), Topi, Pakistan in 2006. He is currently pursuing a master degree in computer and communication security from School of Electrical Engineering & Computer Science (SEECS), Islamabad.

Mr. Shoaib is currently working as a system administrator in data center of a reputable financial institution in Islamabad and heads the information security team. He started his career by working as a software engineer in a software house in Lahore where he worked on several projects including development of a GPS application for Symbian mobiles and a gaming client module. He has attained considerable experience in IT and his primary focus of research is on security.