We suggest an idea to dump executable code from memory for malicious application analysis on Android platform. Malicious applications are getting enhanced in terms of antianalysis techniques. Recently, sophisticated malicious applications have been found, which are not decompiled and debugged by existing analysis tools. It becomes serious threat to services related to embedded devices based on Android. Thus, we have implemented the idea to obtain main code from the memory by modifying a part of Dalvik Virtual Machine of Android. As a result, we have confirmed that the executable code is completely obtainable. In this paper, we introduce the existing analysis techniques for Android application, and antianalysis techniques. We then describe the proposed method with a sample malicious application which has strong antianalysis techniques.

1. Introduction

Information communication technology convergence is bringing new technology and service paradigm, that is, machine-to-machine which allows both wireless and wired systems to communicate with other devices, thus expected to make a big change to our economy and society. For a notable example, the development of mobile ecosystem based on mobile devices which support various sensor features has already improved our lives [1, 2]. However, it raised severe security concerns. A typical example is Android mobile platform. It is becoming a prime target for malicious application makers since Android accounts for a majority of mobile market share, and applications can be easily installed through various routes. Moreover, Android is ported not only to mobile devices but also to diverse embedded devices (e.g., a smart watch with sensors such as accelerometer, gyroscope, and heart rate). For these reasons, the number of malicious applications targeting Android is getting higher [3].

From the perspective of malicious application analyzers, it is required to utilize various tools and techniques to deal with a growing number of malicious applications. For instance, dynamic debugging or decompiling technique is commonly used in order to understand the logic of the malicious applications [4]. It is rather easy to decompile an Android application due to the structural characteristics of Java bytecode [5]. In case of pure executable code which is not obfuscated, it can be converted to almost the same as original code. However, some malicious applications take long time for analysis because of their antianalysis techniques such as antidebugging, antidecompiling, and antitamper which have been originally developed for protecting property of applications [6, 7]. Particularly sophisticated malicious applications have been recently found. They do not allow debugging, decompiling, and repackaging. The existing tools and techniques are not enough to deal with them. In this case, what we can do for analysis within the time limit is just to monitor their behaviors on modified Android platform [8, 9]. However, it allows an analyzer only to know which API is called at runtime [10–12]. Thus, it is difficult to identify malicious behaviors if it has a trigger. We cannot even know the existence of the trigger without static analysis on the executable code or dynamic debugging [13]. Hence, it is required to study new techniques to deal with this kind of malicious applications.

This paper is organized as follows: Section 2 deals with the existing techniques for Android application analysis;
Section 3 describes the antianalysis techniques used for protecting malicious applications; Section 4 introduces the sophisticated malicious application recently found as a sample; Section 5 represents the proposed method to address the current problem; Section 6 shows the results of the implementation as a proof-of-concept. Finally Section 7 concludes the paper.

2. Existing Techniques for Android Application Analysis

An Android application package file (APK) is a ZIP format which basically consists of a single executable file implemented in Java, a configuration file called AndroidManifest.xml, and some resources. The APK can be analyzed statically and dynamically. They have pros and cons [14]. Thus, it is recommended to use both of them to analyze applications especially malwares.

2.1. Static Analysis

Decompiling. It allows an analyzer to comprehend the logic of an application. An Android application is commonly developed in Java language based on Android SDK. After build process, Java bytecode is transformed to Dalvik executable (DEX) format. We can simply get Java source code of an application after extracting the DEX file from a package file. To do this, it is required to convert the DEX format to Java bytecode first, and then it can be easily decompiled by using tools like dex2jar and JAD, respectively. The output of compiling the DEX file is clearer than the output of decompiling binaries (e.g., PE format of Windows or ELF format of Linux) due to the structural characteristics of Java bytecode. If the executable code is not obfuscated, it is possible to get almost the same as original one. In addition, Android provides an application with a capability to use native code using the Java Native Interface (JNI) in form of a library implemented commonly in C language. Application developers can address some issues related to performance and porting by taking an advantage of the capability. In case of the native code, it can be decompiled by Hex-Rays, a plugin of IDA Pro, which is well-known for comprehensive binary analysis tool. It is strongly recommended to decompile executable code to see the logic since the source level analysis enables full browsing to understand inner mechanisms of an application for malware analysis.

Disassembling and Assembling. A disassembler translates executable code to assembly language—the inverse operation to that of an assembler. Baksmlai and Smlai are the representative disassembler and assembler for an Android application [15]. Disassembly shows APIs used in an application but it offers less readability compared to the decompiled output. Disassembling and assembling are also used to convert an ODEX file (optimized DEX file which is dependent on the hardware) to the general DEX format. A DEX file is optimized before being loaded into the memory when executed. As mentioned above, a DEX file is necessary for decompilation. An ODEX file is located in a cache directory of a device, which means it is possible to perform static analysis as long as either a DEX file or an ODEX file is obtained. Figure 1 shows the decompilation process of a general executable file (DEX) and an optimized executable file (ODEX), respectively. However, the most powerful capability of disassembling and assembling is that it allows making modifications to the original code and repackaging an application (further discussed below).

2.2. Dynamic Analysis

Dynamic Debugging. It allows tracing execution flow of an application with a plenty of dynamic information. Dynamic
debugging either Java layer or native layer can be conducted. When we perform dynamic debugging Java layer of an application without source code, it is required to prepare for disassembled code of its DEX file using a disassembler like Baksmali and add a specific API in the main activity of an application in order to make it wait for a debugger. In addition to it, a debuggable attribute of the AndroidManifest.xml file of the target application should be set to true. After making modifications to the disassembled output, it needs to be repackaged. This process can be performed by an automated tool called Android APKTool. At this point, the target application should be signed by APK sign tool; otherwise the modified application cannot be installed on a device. In case of native layer, there is no need to make a modification to the target application. The native layer of the target application can be dynamically debugged by a traditional tool like GDB (GNU Debugger) on the underlying Linux.

**Behavior Monitoring.** It is to observe behaviors of an application after installing an application on the modified Android platform. The modified Android platform is usually designed to log specific messages related to malicious behaviors like a privacy threat in order to provide information for determining malwares. Tracedroid, one of the behavior monitoring systems, is well-known for a free online analysis service [16]. This kind of approach can reduce analysis time since it needs a minimum interaction and effort from an analyzer but the problem is that it may fail to analyze an application if the application has a trigger to activate a malicious code. This is why static analysis needs to be conducted as well as dynamic analysis.

### 3. Antianalysis Techniques

As mentioned in the previous section, it is not difficult to analyze an Android application using some tools. Thus, it caused concerns that it may threaten intellectual property of an application. For this reason, antianalysis techniques have been developed to protect an application. In this section, representative antianalysis techniques are introduced.

#### 3.1. Code Obfuscation.

It makes reverse engineering difficult by transforming a code into a form which looks different and unclear but functionally does the same thing as the original one. It makes an analyzer need much time to understand the logic. There are various techniques for code obfuscation in Java layer such as class and method renaming, control flow manipulation, string encryption, class encryption, and API hiding [6]. Of course, similar techniques are used in native layer.

#### 3.2. Antidecompiling.

It is to manipulate executable code so that a decompiler cannot process it properly. It usually inserts a trash code or changes a data structure skillfully. By doing this, it induces a decompiler to produce a wrong decompiled code or stop a decoding process as soon as it faces unknown code which is manipulated by an antidecompiler. It however lets the modified code be executed at runtime without any problem.

#### 3.3. Antidebugging.

It is to prevent an application (or process) from being debugged. Antidebugging techniques can be classified into two types according to the layer. In order to protect Java layer against a debugger, it is required to check integrity of an application since the target application should be modified for debugging as described in Section 2. This kind of technique to check integrity is called antitamper. Once the antitamper recognizes that it has been modified, it changes a code flow instead of running a main code. On the other hand, debugging the native layer is not required to modify an application. When debugging the native layer, GDB is commonly used to attach and trace the target process on the underlying Linux. There are a variety of techniques for antidebugging. One of them is presented in Section 4.

#### 3.4. Dynamic Class Loading.

Android allows an application to load external code at runtime in some ways. It is used for addressing some limitations; for example, the number of methods that a DEX file can have is up to 64 K, but an application can overcome the limitation by loading additional code with the technique. Note that improper use of the techniques is prone to being attacked like malicious code injection [17]. By the way, some applications use it for the antianalysis regardless of its original intention [15]. A detailed explanation of the technique for the antianalysis is described in Section 4.

### 4. Sophisticated Malicious Application

We have recently found a sophisticated malicious application spreading via Smishing. It has complicated antianalysis techniques to protect its main code. As a result of conducting analysis at VirusTotal which is a free online malware scanner service, 16 out of 53 antivirus products could identify it as a malware as of May 14, 2014. It is identified as a keyword Trojan-Banker by some of the antivirus products [18]. We therefore have named the malicious application Trojan-Banker in this paper. It is very hard to analyze the Trojan-Banker by using the existing techniques and tools. In this section, we describe the antianalysis techniques applied to the Trojan-Banker.

There are four files which are important in the Trojan-Banker package in terms of analysis—classes.dex that is a default executable file, AndroidManifest.xml that is a kind of configuration file, libsec.so that is a native library developed in C/C++, and external.jar which is an extra file as shown in Figure 2.

The top and the bottom are default files for an Android application, and the two in the middle are additional files to which we need to pay attention. We can know what kind of components that an application has by examining AndroidManifest.xml since every component should be defined in AndroidManifest.xml on the development stage [19, 20]. As a result, we have confirmed 34 activities, 3 services, 1 receiver, and 19 permissions. Box 1 shows some of the activity names defined in the AndroidManifest.xml file of the sample application. There exist some major bank names of South Korea such as wooribank, kbbank, and nhbank. Thus, we can
Box 1: A list of activity components in the AndroidManifest.xml file of the sample application.

Figure 2: Overview of Trojan-Banker.

assume that the sample is related to financial malware. In order to be sure, we should analyze the executable file to find malicious code.

The classes.dex which is the default executable file can be decompiled by existing tools. However, there is one receiver and a few custom classes only in classes.dex file although AndroidManifest.xml says that it has rich components and a variety of permissions.

All the classes.dex does is to import a native library file and invoke DexClassLoader API with a specific file (external.jar) that is saved in the application package. The DexClassLoader API provided by Application Framework Layer
Table 1: Function call sequence for DexClassLoader.

| Type | Path | File | Method (function) |
|------|------|------|-------------------|
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexClassLoader.java | DexClassLoader |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | BaseDexClassLoader.java | BaseDexClassLoader |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexPathList.java | DexPathList |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexPathList.java | loadDexFile |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexFile.java | loadDex |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexFile.java | DexFile |
| Java | libcore/dalvik/src/main/java/dalvik/system/ | DexFile.java | openDexFile |
| CPP  | dalvik/vm/native/ | dalvik_system_DexFile.cpp | Dalvik_dalvik_system_DexFile_openDexFile |
| CPP  | dalvik/vm/ | JarFile.cpp | dvmJarFileOpen |
| CPP  | dalvik/vm/ | DvmDex.cpp | dvmDexFileOpenFromFD |

is for dynamically loading an external executable file [21], which means that the external executable file (external.jar) may have the rest of the components. But the external.jar file seems to be encrypted. It has no any other magic number and the entropy is high as shown in Figure 2. Hence, we can have an assumption that the native library has features to decrypt the external.jar file at runtime but unfortunately the library file is highly obfuscated with various techniques.

One of the distinctive features that the native library provides is an antidebugging technique. The Trojan-Banker does not allow debugging by means of taking advantage of a fork-attach technique that a child process preempts its parent process at native layer. In addition, it terminates its own process as soon as it detects either it is tampered or it is run on inappropriate device environment like emulator in order to prevent it from being analyzed.

To sum up, it is required to analyze the native library first to fully understand the mechanism of antianalysis techniques to detour them in order to reach the main code of the external.jar file. It however seems to be a very tough job because of the obstacles such as antidebugging, antitamper, and obfuscation techniques. The only thing which is allowed with the existing tools and methods is just to observe behaviors of the target application on the modified platform like Tracedroid unless the native library is thoroughly examined but it would fail to determine malicious behaviors if it has a trigger as mentioned in Section 2. Therefore, it is essentially required to get decrypted code of the external.jar file for decompilation, which only makes fully understanding the logic possible.

5. Proposed Method

The idea that we suggest is to extract executable code from the memory at runtime rather than analyzing the native library to understand the tricky antianalysis techniques since every executable file is loaded into the memory when executed. There are two similar approaches in terms of extracting code from the memory.

The first one is called AndroDump which is a part of AndroGuard—a comprehensive reverse engineering tool for Android application [22], but it cannot be used for solving the current issue because it is just to find executable code by searching the magic number of Java class file in memory at a given time and dump them after attaching the target process. The problem of the AndroDump is that it cannot even attach the sample application due to its antidebugging technique.

The other one is proposed by Sophos Lab—international security company which is well-known for antivirus products. The solution of Sophos Lab is to seek executable code with its structure data from the whole memory dump file by using a plugin of the volatility which is a single and cohesive framework for memory analysis of multiplatforms. To do this, Sophos Lab uses Linux Kernel Module (LKM), named LiME (Linux Memory Extractor) that is used for acquisition of volatile memory [23]. Of course, dumping whole memory is not affected by antianalysis techniques of the sample since it is conducted at kernel level which is lower than the layer the sample application works on. However, the critical problem of this solution is that it takes a long time to acquire whole memory of the system. Moreover, it cannot guarantee the desired executable code still remains in the memory while the memory dumping is being performed because the target application may terminate itself as soon as it finds something wrong in execution environments to avoid being analyzed.

Therefore, a novel approach is required to address this issue, which can catch the moment the executable code is loaded into the memory. In addition, it should be light enough to use practically. For satisfying these requirements, we suggest an idea to dump executable code in Dalvik Virtual Machine (DVM) of Android platform because every executable code written in Java is handled in the DVM. There have not been any approaches dealing with the DVM to solve the problem. Firstly, we analyzed the mechanism of DexClassLoader of the DVM based on Android platform source version 4.0.1. As a result, we could understand the process of loading external executable files. Table 1 presents function call sequence.

When an application invokes DexClassLoader API provided by the Android SDK, it finally calls dvmDexFileOpenFromFd at native code layer of Dalvik Virtual Machine. The dvmDexFileOpenFromFd function loads optimized executable code to the memory, which is performed after the decryption routine of the native library (libsec.so) that we are not interested in analyzing. We can find a specific address into
which the executable code is loaded at the dvmDexFileOpenFromFd function. Thus, it is expected that we would be able to extract the optimized executable code if we insert dump-code into the function.

At first, we just tried to extract every optimized executable code which is loaded by the dvmDexFileOpenFromFd function but we faced an exception error that occurs in the child process of the malicious application right after the dvmDexFileOpenFromFd function is invoked by the child process. In general, applications load some core executable files of system as well as their executable files but the problem is that the target process is killed as soon as the dump-code tries to access memory of the child process, which is not yet determined. Further analysis is required to know the cause.

However, it is not necessary to address the issue now because what we need is the optimized executable file which is loaded only by the main process of the malicious application not the child process. The child process exists only for antidebugging. Thus, we need to recognize the target process at the dvmDexFileOpenFromFd function in order to prevent it from being killed and dumping unnecessary executable files that we do not need. For these reasons, we have designed the implementation of the dump-code for proof-of-concept as shown in Figure 3. This is a small modification to the DVM of Android platform.

When dvmDexFileOpenFromFd function is invoked, the dump-code works as follows:

1. Read user-defined process name from a dump.conf file in flash memory.
2. Get current pid and parent pid using getpid(), getppid() functions.
3. Read a cmdline file using obtained pid since the process name is saved in the cmdline.
4. Compare the current process name to the user-defined process name. If it is matched, then check whether its parent process name is zygote because every application is forked from the zygote process, which means that we can identify the child process of the malicious application since the parent process of the child process is not zygote process.
5. Dump the optimized executable code from the memory to the specific directory in the flash memory.

6. Results and Evaluation

We evaluated the proposed method with the sample application. The evaluation procedures are as follows. We will extract all the executable codes dynamically loaded into the memory by the sample application. Of all the extracted files, we will identify the main executable file that was originally encrypted in the application package. And then, we will convert its format to DEX for decompilation. The next step is to find the specific activity names in the decompiled code to see if the names mentioned in Box 1 really exist. If they exist with their code, it means that the proposed method works properly.

For the evaluation, we firstly need to know the process name of the sample application to designate the target in the dump.conf file as described in Figure 3. We can simply get the process name via various ways (e.g., examining the AndroidManifest.xml file, invoking API of PackageManager class provided by Android SDK, or using system command like “ps” on the Linux). Among them, Figure 4 shows one of the ways to check the process name by examining the AndroidManifest.xml file.

The target process name is com.madabai.kim. After designating the target process name in the dump.conf file as described in Figure 3, we will extract all the executable files dynamically loaded into the memory by the sample application. Of all the extracted files, we will identify the main executable file that was originally encrypted in the application package. And then, we will convert its format to DEX for decompilation. The next step is to find the specific activity names in the decompiled code to see if the names mentioned in Box 1 really exist. If they exist with their code, it means that the proposed method works properly.

For the evaluation, we firstly need to know the process name of the sample application to designate the target in the dump.conf file as described in Figure 3. We can simply get the process name via various ways (e.g., examining the AndroidManifest.xml file, invoking API of PackageManager class provided by Android SDK, or using system command like “ps” on the Linux). Among them, Figure 4 shows one of the ways to check the process name by examining the AndroidManifest.xml file.

The target process name is com.madabai.kim. After designating the target process name in the dump.conf file, all we have to do is just to install and run the target application on the modified platform. As a result of running it, we can see that there are three files extracted from the memory (see Figure 5). The file name consists of the process name and its file size.

We added some code to the DVM to leave a message related to dynamic code loading. Thus, we can get information about extracted files through log messages at LogCat as shown in Figure 6. The first one whose size is 168528 bytes is an executable file in android.test.runner.jar which is not interesting. The second one whose size is 20088 bytes is the default executable file of the sample application. The last one whose size is 2265048 is the external executable file located in the cache directory of the sample application. This
is the external.jar which is loaded dynamically after being decrypted by the native library.

The extracted executable file is ODEX format as shown in Figure 7. Thus, we need to convert it to DEX format first as mentioned in Section 2. The Baks, Smali & Smali tools can be used for it.

Figure 8 represents the result of decompiling the converted file using the JEB which is a powerful interactive Android application decompiler [24]. Fortunately, the external file is not obfuscated. We can therefore see most of the components that are declared in the AndroidManifest.xml in a clear state. There exist the activities whose names are the major banks of South Korea in the left of Figure 8. They are the same as shown in Box 1. After a short analysis of the decompiled code (main code), it turned out that it shows fake banking screens and collects financial information from user as shown in the right.

Figure 9 also shows the remote server address which is supposed to collect financial information but the address is not available at the moment.
7. Conclusion

The number of malicious applications targeting Android is getting higher. It is also getting harder to analyze them due to their advanced antianalysis techniques. Recently, sophisticated malicious applications have been found, which hide their main code with encryption and some advanced techniques. In this case, it is a difficult and time-consuming task to analyze them by using existing tools. Therefore, we suggest an idea to extract the main executable code from the memory at Dalvik Virtual Machine (DVM) in order to get source code for providing quick and efficient analysis environment. It is
a robust solution for the malicious applications that hide their executable code. This is the first try to deal with the DVM to address the issue. We opened the door to those who are working on Android application analysis to research further work related to the DVM. We expect that it would be very helpful to provide a code extraction service for malicious application analyzers on the online. We also anticipate that malware makers will focus on obfuscation techniques in the future since the techniques for hiding executable code are no longer valid.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the ICT R&D program of MSIP/IITP, Republic of Korea (13-912-06-003, Development of Mobile S/W Security Testing Tools for Detecting New Vulnerabilities of Android). This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2015-H8501-15-1008) supervised by the IITP (Institute for Information & Communications Technology Promotion).

References

[1] M. Kuna, H. Kolaric, I. Bojic, M. Kusek, and G. Jezic, "Android/OSGi-based Machine-to-Machine context-aware system," in Proceedings of the 11th International Conference on Telecommunications (ConTEL’11), pp. 95–102, IEEE, June 2011.
[2] C. Tucker, R. Tucker, and J. Zheng, "StopWatcher: a mobile application to improve stop sign awareness for driving safety," International Journal of Vehicular Technology, vol. 2012, Article ID 532568, 8 pages, 2012.
[3] D. Kim and J. Ryou, "SecureSMS: prevention of SMS interception on android platform," in Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication (ICUIMC ’14), ACM, January 2014.
[4] M. Louk, H. Lim, and H. Lee, "An analysis of security system for intrusion in smartphone environment," The Scientific World Journal, vol. 2014, Article ID 983901, 12 pages, 2014.
[5] G. Nolan, Decompiling Android, Apress, New York, NY, USA, 2012.
[6] Y. Piao, J. H. Jung, and J. H. Yi, "Deobfuscation analysis of DexGuard code obfuscation tool," in Proceedings of the International Conference on Computer Applications and Information Processing Technology, pp. 107–110, 2013.
[7] H. L. Thanh, "Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it," Journal of Information Security, vol. 4, no. 4, pp. 213–224, 2013.
[8] W. Enck, P. Gilbert, B.-G. Chun et al., "Taint droid: an information flow tracking system for real-time privacy monitoring on smartphones," Communications of the ACM, vol. 57, no. 3, pp. 99–106, 2014.
[9] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, “MAD-AM: a multi-level anomaly detector for android malware,” in Computer Network Security, vol. 7531 of Lecture Notes in Computer Science, pp. 240–253, Springer, Berlin, Germany, 2012.
[10] Y. Liu, S. Jia, and C. Xing, “A novel behavior-based virus detection method for smart mobile terminals,” Discrete Dynamics in Nature and Society, vol. 2012, Article ID 262193, 12 pages, 2012.
[11] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “Andromaly: a behavioral malware detection framework for android devices,” Journal of Intelligent Information Systems, vol. 38, no. 1, pp. 161–190, 2012.
[12] M. Zhao, T. Zhang, F. Ge, and Z. Yuan, “RobotDroid: a light-weight malware detection framework on smartphones," Journal of Networks, vol. 7, no. 4, pp. 715–722, 2012.
[13] C. Wang, Z. Wu, X. Li, X. Zhou, A. Wang, and P. C. Hung, “SmartMal: a service-oriented behavioral malware detection framework for mobile devices," The Scientific World Journal, vol. 2014, Article ID 109866, 11 pages, 2014.
[14] S.-H. Ju, H.-S. Seo, and J. Kwak, “Study on analysis methodology for android applications," Journal of Internet Technology, vol. 14, no. 5, pp. 851–858, 2013.
[15] Y. Zhou and X. Jiang, “Dissecting Android malware: characterization and evolution," in Proceedings of the 33rd IEEE Symposium on Security and Privacy, pp. 95–109, May 2012.
[16] VU University Amsterdam, “Tracedroid,” http://tracedroid.fuw.vu.nl.
[17] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute this! Analyzing unsafe and malicious dynamic code loading in android applications," in Proceedings of the Network and Distributed System Security Symposium, San Diego, Calif, USA, February 2014.
[18] VirusTotal-Trojan_Banker, 2014, https://www.virustotal.com/en/file/44fd78b61ee1585609f7623a92b3fb1a341d3e090d3bc5c91ccad-e2d2fe48a542/analysis/.
[19] Z. Aung and W. Zaw, "Permission-based android Malware detection," International Journal of Scientific & Technology Research, vol. 2, no. 3, pp. 228–234, 2013.
[20] T. Vidas, C. Zhang, and N. Christin, “Toward a general collection methodology for Android devices," Digital Investigation, vol. 8, pp. S14–S25, 2011.
[21] "Android DexClassLoader API," http://developer.android.com/reference/dalvik/system/DexClassLoader.html.
[22] A. Desnos, “Androguard—AndroDump,” 2014, http://code.google.com/p/androguard/source/browse/androdump.py.
[23] R. Yu, "Android packers: facing the challenges, building solutions," in Proceedings of the Virus Bulletin Conference (VB ’14), pp. 266–275, Seattle, Wash, USA, September 2014.
[24] JEB Decompiler, https://www.pnfssoftware.com/.
Submit your manuscripts at http://www.hindawi.com