Survey of Dynamic Anti-Analysis Schemes for Mobile Malware

Jongsu Lim, Yonggu Shin, Sunjun Lee, Kyuho Kim, and Jeong Hyun Yi*
School of Software, Soongsil University, 06978, Republic of Korea
{jongsu253, tls09611, starj1024, krbgh205760}@gmail.com, jhyi@ssu.ac.kr

Abstract

With the development of the smartphone market, the smartphone application market will grow significantly. As a result, malicious code targeting smartphones is increasing exponentially. Attackers are spreading malicious apps by embedding malicious code in the app through repackaging attacks. Small-scale payment fraud and malicious files for smart banking also skyrocketed through smishing attack targeting Android smartphone users. The intelligent attack, which refers to the type of target attack, has also become fully visible. In the future, mobile payment and electronic financial crime targeting smartphone users are expected to become more popular through malicious files based on Android spreading through smishing attack and it is predicted that various irregular mobile security threats will come true. To prepare for such attacks, several analytical tools have been developed, including a sandbox tool that can analyze Android malicious apps. However, as in PC environment, we anticipate the emergence of anti-analysis schemes that can neutralize these analytical tools. Therefore, this paper analyzes the anti-analysis schemes applied to malicious applications. By supporting the analysis of malicious applications based on the results of this work, it will be very helpful to reduce the research cost of malicious code research and to create a secure smartphone security environment.

Keywords: Anti-rooting, anti-emulating, anti-debugging, mobile code reversing

1 Introduction

For mobile applications based on Android, which have the highest share in the mobile market [1], decompiling is easily available and repackaging attacks [2] are especially common because of the use of intermediate language, `Dalvik bytecode`, which is a new reconstruction of Java bytecode according to Android's own method. It inherits the characteristics of existing Java language and has a lot of symbol information. Since it has an intuitive instruction set system, it is fundamentally vulnerable to decompiling. These features can also be targeted to malicious applications, such as *static analysis* that analyzes source code by decompiling an application suspected of malicious activity, or *dynamic analysis* that observes and analyzes actual application execution time. It is possible to distinguish malicious or not.

As a result, mobile malware developers are adopting anti-analysis schemes to protect their applications from analysis, and their technology is becoming more and more sophisticated. Mobile malware with anti-analysis schemes can delay the analysis period from a few days to a few months for as long as the delay period. Damage can be one person, and it can affect groups or countries, so it is necessary to respond quickly. Therefore, in this paper, we report the results of analyzing the structures of anti-analysis schemes applied to mobile malware. Based on this analysis, it is expected that the analysis of mobile malware with anti-analysis scheme will be proceeded smoothly, thereby ensuring rapid response and minimizing damage, thereby providing a reliable and secure mobile environment.

The paper is organized as follows. We classify anti-analysis schemes in Section 2. Section 3 presents the analysis results for anti-rooting schemes, Section 4 deals with anti-emulating schemes, and Section 5 describes anti-debugging schemes. Finally, Section 6 concludes.

*Corresponding author: School of Software, Soongsil University, Seoul, 06978, Korea, Tel: +82-2-820-0914

2 Classification of Anti-Analysis Schemes

Anti-analysis scheme is a countermeasure to prevent reverse engineering analysis of codes. As shown in Table 1, *static anti-analysis* scheme and *dynamic anti-analysis* scheme can be classified according to the timing of the analysis scheme. Static anti-analysis schemes include *obfuscation* [3], *packing* [4], and *tamper detection* [5]. Dynamic anti-analysis schemes include *anti-rooting* [6], *anti-emulating* [7], and *anti-debugging* [8]. This paper focuses on dynamic anti-analysis schemes applicable to the Android environment.

Table 1: Classification of Mobile Anti-Analysis Schemes

Schemes		Description
Static	`Obfuscation`	A technique that makes it difficult to analyze code by changing some or all of the application executable files
	`Packing`	A technique for compressing and hiding the executables to avoid exposing the original executables during static analysis
	`Tamper Detection`	A technique for judging whether or not the executable file is forged using the integrity checking scheme
Dynamic	`Anti-Rooting`	A technique for detecting whether a user or an application can be granted the highest privilege on the device where the application is running
	`Anti-Emulating`	A technique that detects that the device on which the application is running is operating in an emulator environment that is not a real device but a virtual configuration
	`Anti-Debugging`	A technique to detect that the application's own execution flow is being analyzed by another process

3 Anti-Rooting Schemes

The *anti-rooting* scheme is a technique for determining whether the device is rooted by checking the changed system properties due to rooting or checking whether the binary files and applications related to the rooting are installed.

3.1 Application Package Checking

To root the Android device, it is necessary to have applications like `SuperSu.apk`, `KingRoot.apk`, `towelroot.apk` installed. As shown in Figure 1, on Android, the `getInstallPackages()` method of `PackageManager` allows you to see a list of applications installed on the device, which can detect the presence of the above-mentioned rooting related package to detect the rooting.

3.2 Binary File Checking

Rooted Android devices have a `su` binary file in the directory that stores the executables, such as `/system/bin/`, `/system/xbin/`, and `/sbin/`. As shown in Figure 2, it is possible to obtain the root privilege by executing the `su` binary file existing in the corresponding path, so that the rooting can be detected by checking if the `su` binary file exists in the corresponding path.

```
checkPackages() - Java pseudocode

checkPackages() {
   PackageManager pm = context.getPackageManager();
   List<PackageInfo> packages = pm.getInstalledPackages(
      PackageManager.PerMISSION_GRANTED);
   for(PackageInfo package : packages) {
      if(package.packageName.equals("com.noshufou.android.su")) {
         // Detect Rooting
      }
   }
}
```

Figure 1: Sample Code for Checking Binary Files Related to Rooting

```
checkBinary() - C pseudocode

checkBinary() {
   char *su_path[] = { "/system/bin/su", "/system/xbin/su",
                       "/sbin/su", ... };
   struct stat st;
   for(i = 0 ; i < su_path_num ; i++) {
      if( stat(su_path[i], &st) == 0) {
         // Detect Rooting
      }
   }
}
```

Figure 2: Sample Code for Checking Binary Files Related to Rooting

3.3 System Property Checking

The Android system has a `build.prop` file that shows the properties of the built system image. The `build.prop` file contains various system property information such as `sdk` version information, release version, product name, and so on. Among the property information stored in the `build.prop` file is the property `ro.build.tags` which describes the build. For an officially deployed Android image, the `ro.build.tags` property is set to `release-keys`, but for images used in an unofficially built image or emulator, the `ro.build.tags` property is set to `test-keys`. It can also be set to various values such as `unsigned`, `debug` depending on the build environment.

Anti-rooting through system attribute detection is a technique to detect unofficially deployed Android images and finds that the `ro.build.tags` attribute is not set to `release-keys` as shown in Figure 3 and Table 2. This is a detection technique that assumes that an Android image that is not formally released will be rooted, so a device that is not actually rooted can be detected as a rooted device.

```
checkProperty() - Java pseudocode
checkProperty() {
   String build_tags = SystemProperties.get("ro.build.tags");
   if( build_tags.contains("release-keys") != false ) {
      // Detect Rooting
   }
}
```

Figure 3: Sample Code for Checking System Property Related to Rooting

Table 2: Comparison of System Property Between Real Device and Rooted Device

Real Device	Rooted Device
ro.build.version.release=4.4.4	ro.build.version.release=4.4.4
ro.build.date=Fri Jun 13 07:05:49 UTC 2014	ro.build.date=Thu Feb 19 02:20:50 UTC 2015
ro.build.date.utc=1402643149	ro.build.date.utc=1424312450
ro.build.type=user	ro.build.type=eng
ro.build.user=android-build	ro.build.user=android-build
ro.build.host=kpfj3.cbf.corp.google.com	ro.build.host=vpbs13.mtv.corp.google.com
ro.build.tags=release-keys	**ro.build.tags=test-keys**

3.4 Process List Checking

On the Android system, there is a `proc` virtual file system that allows you to view information about running processes. The `proc` virtual file system is a virtual file system that provides information about the processes managed by the kernel in the user area. In the `proc` virtual file system, we can find various information such as the process name, the parent process ID, and the memory space being used.

As shown in Figure 4, the anti-rooting scheme using the process list is a technique for checking the name of the running processes provided by the `proc` virtual file system to check whether the process related to the rooting is being executed. That is, it is a technique to detect that a rooting binary file or an application that is not detected by the above-described anti-rooting schemes is operating. As shown in Figure 5, the name of the process running in the `proc` virtual file system can be found in `/proc/[pid]/cmdline`.

4 Anti-Emulating Schemes

The emulating environment has unique properties of the emulator and unique modules added to run the emulator on the host PC without any problems [9]. The *anti-emulating* scheme is a technique to detect the emulator by using the unique information of the emulator.

```
checkProcList() - C pseudocode
```

```
checkProcList() {
   DIR *dir; struct dirent *dir_entry; char cmdline[128];
   int fd; char buf[128];
   dir = opendir("/proc");
   while( dir_entry = readdir(dir) ) {
      sprintf(cmdline, "/proc/%d/cmdline", dir_entry->d_name);
      fd = open(cmdline, O_RDONLY);
      read(fd, buf, 128);
      if( !strcmp(buf, "su") | !strcmp(buf, "/system/bin/sh") | ... ) {
         // Detect Rooting
      }
   }
}
```

Figure 4: Sample Code for Checking Processes Related to Rooting

```
shell@generic:/ # cat /proc/1/cmdline : echo
/init
shell@generic:/ #
```

Figure 5: Results from /proc/[pid]/cmdline

Table 3: Comparison of Device Property Between Real Device and Emulator

Properties	Real Device (Nexus S)	Emulator
IMEI	356951040948493	0
Line 1 Number	0	15555215554
Network Operator	45008	310260
Sim Operator	45008	310260
Sim Operator Name	KT	Android
SubscriberID	450084510014409	310260000000000
Voice MailNumber	Null	15552175049
Board	herring	unknown
Brand	Google	generic
Manufacturer	Samsung	unknown
Model	Nexus S	sdk
Product	sojuk	sdk
Serial	34308265ACC200EC	unknown

4.1 Device Property Checking

The Android device stores device attribute information such as device unique number (IMEI), telephone number, manufacturer, and model name. The attribute information of the device is set by a manufacturer, a mobile communication company, and SDK developers, which is set to identify the device. However, since the Android system image used in the emulator is not an image generated by a separate maker or a communication company, the device information is set as default information set in the Android open source. As shown in Table 3, the emulator can be detected when comparing the attribute information of the currently running device with the device setting information of the emulator image.

4.2 Exclusively Used Process Checking

The emulator virtualizes the execution environment to provide the same execution environment as the actual device. In this process, interfacing problems with hardware, host OS, and virtual environment occur. Fixed information such as device driver and library to solve this problem is included. As shown in Figure 6, it is possible to detect the emulator using this information.

```
shell@generic:/ # ls -Ral dev | grep qemu
crw-rw-rw- system system 10, 62 2016-07-25 03:29 qemu_pipe
srw-rw-rw- root   root          2016-07-25 03:29 qemud
```

Figure 6: Checking Processes Exclusively Used by Emulator

4.3 Kernel Log Checking

The device driver added to solve the interface problem in the emulator environment outputs its operation log through the `kernel log` of the OS operating in the virtual environment. In the output `kernel log`, the signature is left as the name of the emulator or the code name, which can detect the emulator environment as shown in Figure 7.

```
shell@generic:/ # dmesg | grep qemu
<5>Kernel command line: qemu.gles=0 qemu=1 console=ttyS0 android.qemud=ttyS1
                     androidboot.hardware=goldfish android.checkjni=1 ndns=2
<4>goldfish_new_pdev qemu_pipe at ff018000 irq 19
```

Figure 7: Detecting Emulator by Checking Kernel Logs

5 Anti-Debugging Schemes

The *anti-debugging* scheme is a technique for detecting the debugger based on the state information that is changed during application debugging or based on the structural characteristics of the debugging process [10].

5.1 `TracerPID` Checking

To debug processes on a Linux-based system, you must use the `ptrace` system call provided by the Linux kernel. The `ptrace` system call allows you to gain control over other processes. In effect, the kernel is running to control the process, setting debugging-related information among the process state information managed by the kernel. That is, the debugger can be detected by checking the set debugging related information. The status information of the processes managed by the kernel is provided in the `/proc` virtual file system so that it can be viewed in the user area via `/proc/[pid]/status`. In `/proc/[PID]/status`, `status` information such as process name, operation status, and PID of process can be checked. Among the status information, PID information of the process debugging process in which `TracerPID` item is running is provided. The `TracerPID` entry is set to `0` if the process is normally running, but it is set to the PID of the debugger process if it is being debugged. By checking this as shown in Table 4, the debugger can verify that the process is being debugged.

Table 4: Detecting Debugger by Checking `TracerPid`

Normal Process	**Debugging Process**
`$ cat /proc/1754/status`	`$ cat /proc/1754/status`
`Name:    m.android.email`	`Name:    m.android.email`
`State:   S (sleeping)`	`State:   S (sleeping)`
`Tgid:    1754`	`Tgid:    1754`
`Pid:1754`	`Pid:1754`
`PPid:    201`	`PPid:    201`
`TracerPid:   0`	**`TracerPid:   7270`**
`Uid:10027    10027    10027    10027`	`Uid:10027    10027    10027    10027`
`...`	`...`

5.2 Debugging API Checking

As mentioned earlier, the `ptrace` system call, which provides debugging facilities on Linux systems, provides process control. The `ptrace` system call provides various control functions according to the argument value passed as the request argument. The argument can be set to the request argument value such as control request for other process, memory R/W, system call hooking, signal hooking, etc. The `PTRACE_TRACEME` request parameter value can check whether or not the process itself is being debugged (See Figure 8).

5.3 Timing Checking

The timing check scheme is based on the execution time of a process. In the case of a process being debugged, the execution of the process is performed through the interaction with the analyzer using the debugger, thereby causing the waiting time for the execution of the process. The execution wait time may be several seconds or several minutes. However, in a typical execution environment, even if latency occurs, it takes less time than msec. This difference can be used to detect that a process is being debugged as shown in Figure 9.

```
detect_debugger() - C pseudocode

detect_debugger() {
   if(ptrace(PTRACE_TRACEME, 0, 0, 0) != 0) {
      // Detect Debugger
   }
}
```

Figure 8: Detecting Debugger by Checking API

```
timing_check() - C pseudocode

timing_check() {
   struct timeval start, end;
   gettimeofday(&start, NULL);
   /*
   * Target code to protect
   */
   gettimeofday(&end, NULL);
   if( end.tv_sec - start.tv_sec < threshold )
      // Detect Debugger
}
```

Figure 9: Detecting Debugger through Timing Checking

5.4 Breakpoint Instruction Checking

The debugger can set breakpoints in the program's instructions to control the execution of the program. If a breakpoint is set, the program will stop running just before executing the command. The principle of stopping the program is to change the command at the point where the breakpoint is set to a command that can not be interpreted by the CPU, thereby stopping the execution of the program from the CPU. Because of this execution control principle, the program can detect the behavior of the debugger. Since the code area of the program is fixed and unchanged after compilation, a unique hash value is generated when a hash algorithm is applied. If the code area changes due to the breakpoint setting as above, reapplying the hash algorithm to the code area generates a new hash value different from the existing hash value. Therefore, it is possible to detect the behavior of the debugger by comparing two hash values.

5.5 Signal Checking

All signals passed to the debugging process are passed to the debugger first, and control over the signal is transferred to the debugger. The debugger can determine whether or not to pass the signal back to the debugging process, but the default option is not to be passed. Based on this principle, if the signal is not delivered to the process after generating the signal, it can be confirmed that the debugger is operating as shown in Figure 10.

```
signal_check() - C pseudocode

int detect = 0;
void signal_handler(int signo){
   detect++;
}
int antiDebugging(){
   detect = 0;
   SIGNAL(SIGINT, signal_handler);
   SIGNAL(SIGINT, signal_handler);
   if( detect != 2 )  {
      // Detect Debugger
   }
}
```

Figure 10: Detecting Debugger by Checking Signal

6 Conclusion

The latest mobile malware contains a variety of anti-analysis schemes, so it can not easily be analyzed with existing malware analysis tools. Also, even if analysis is possible, it takes a lot of analysis time and causes more damage than during the delayed period. In this paper, we have analyzed the structure of anti-analysis schemes applied to mobile malware. Based on this, we will be able to automatically identify anti-analysis schemes rather than passive analysis that relies on reverse engineering analysts in the future and write code that bypasses them in memory. If such an automated anti-analysis evading technology is developed, it will respond quickly to intelligent mobile malware and contribute to minimizing the damage.

Acknowledgments

This research was supported by the Mid-Career Researcher program through the National Research Foundation of Korea (NRF) funded by the MSIT (Ministry of Science and ICT) (NRF-2017R1A2B4008822).

References

[1] R. Molla, "Closing the books on microsoft's windows phone," https://www.recode.net/2017/7/17/15984222/microsoft-windows-phone-mobile-operating-system-android-iphone-ios [Online; accessed on August 1, 2018], July 2017.

[2] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi, "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, vol. 73, no. 4, pp. 1421–1437, December 2013.

[3] W. Yoo, M. Ji, M. Kang, and J. H. Yi, "String deobfuscation scheme based on dynamic code extraction for mobile malwares," *IT Convergence Practice*, vol. 4, no. 2, pp. 1–8, June 2016.

[4] J. Lim and J. H. Yi, "Structural analysis of packing schemes for extracting hidden codes in mobile malware," *EURASIP Journal on Wireless Communications and Networking*, vol. 72, no. 9, p. 221, September 2016.

[5] H. Cho, J. Bang, M. Ji, and J. H. Yi, “Mobile application tamper detection scheme using dynamic code injection against repackaging attacks,” *Journal of Supercomputing*, vol. 72, no. 9, pp. 3629–3645, September 2016.

[6] S.-T. Sun, A. Cuadros, and K. Beznosov, “Android rooting: Methods, detection, and evasion,” in *Proc. of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM’15), Denver, Colorado, USA*. ACM Press, October 2015, pp. 3–14.

[7] T. Vidas and N. Christin, “Evading android runtime analysis via sandbox detection,” in *Proc. of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS’14), Kyoto, Japan*. ACM Press, June 2014, pp. 447–458.

[8] M. N. Gagnon, S. Taylor, and A. K. Ghosh, “Software protection through anti-debugging,” *IEEE Security & Privacy*, vol. 5, no. 3, pp. 82–84, June 2007.

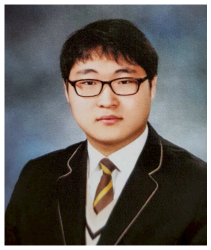
[9] H. Cho, H. Kim, J. Lim, J. Lee, and J. H. Yi, “Empirical analysis of anti-reversing schemes for protecting mobile codes in the internet-of-things,” *International Journal of Services Technology and Management*, vol. 23, no. 1/2, pp. 21–31, January 2017.

[10] M. Schallner, “Beginners guide to basic linux anti anti debugging techniques,” http://www.stonedcoder.org/kd/lib/14-61-1-PB.pdf [Online; accessed on August 1, 2018], May 2006.

Author Biography

Jongsu Lim received the B.S and M.S degrees in Computer Science and Engineering from Soongsil University in 2016 and 2018, respectively. Currently he is a research staff in Cyber Security Research Center. His research interests include binary analysis, reverse engineering, and mobile security.

Yonggu Shin is currently taking a bachelor’s course at School of Software, Soongsil University. His research interests include binary analysis, reverse engineering, and system software security.



Sungjun Lee is currently taking a bachelor’s course at School of Software, Soongsil University. His research interests include binary analysis, reverse engineering, and system software security.

Kyuho Kim is currently taking a bachelor's course at School of Software, Soongsil University. His research interests include binary analysis, reverse engineering, and system software security.

Jeong Hyun Yi is an Associate Professor in the School of Software and the Director of Cyber Security Research Center at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, IoT security, and applied cryptography.