Empirical analysis of anti-reversing schemes for protecting mobile codes in the internet-of-things

Haehyun Cho, Hyunki Kim, Jongsu Lim and Junghwan Lee

School of Computer Science and Engineering,
Soongsil University,
369 Sangdo-ro, Dongjak-gu,
Seoul, 156-743, South Korea
Email: haehyuncho@gmail.com
Email: hitechnet92@gmail.com
Email: jongsu253@gmail.com
Email: jsdksajdlk@gmail.com

Jeong Hyun Yi*

School of Software,
Soongsil University,
369 Sangdo-ro, Dongjak-gu,
Seoul, 156-743, South Korea
Email: jhyi@ssu.ac.kr
*Corresponding author

Abstract: Java-based Android apps are primarily composed of managed code. Managed codes can be easily modified; therefore many static prevention techniques are applied. However, static prevention techniques can be immobilised by dynamic reverse engineering tools. Reverse engineering tools for such managed code operate using QEMU-based emulator methods. Among the many anti-reversing techniques to detect tampering of the application, schemes that terminate the application when an emulator has been detected are being used. In this paper, we compare and analyse the characteristics of the various schemes used to detect emulator-based reverse engineering tools and report experimental results on the effectiveness of the methods in question.

Keywords: anti-reversing; Android APP protection; detecting emulator; protecting mobile codes; internet-of-things.

Reference to this paper should be made as follows: Cho, H., Kim, H., Lim, J., Lee, J. and Yi, J.H. (2017) ‘empirical analysis of anti-reversing schemes for protecting mobile codes in the internet-of-things’, *Int. J. Services Technology and Management*, Vol. 23, Nos. 1/2, pp.21–31.

Biographical notes: Haehyun Cho received his BS and MS degrees in Computer Science from Soongsil University, Seoul, Korea, in 2013 and 2015, respectively. His research interests include mobile application security and mobile platform security.

Hyunki Kim received his BS degree in Computer Science from Soongsil University, Seoul, Korea, in 2015. He is a MS student at School of Computer Science, Soongsil University. His research interests include mobile application security and mobile platform security.

Jongsu Lim is a BS student at School of Computer Science, Soongsil University, Seoul, Korea. His research interests include mobile application security and mobile platform security.

Junghwan Lee is a BS student at School of Computer Science, Soongsil University, Seoul, Korea. His research interests include mobile application security and mobile platform security.

Jeong Hyun Yi is an Associate Professor at the School of Computer Science and Engineering and the Director of the Mobile Security Research Center (MSEC) at Soongsil University, Seoul, Korea. He received his BS and MS degrees in Computer Science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and a PhD degree in Information and Computer Science from the University of California, Irvine, in 2005. He was the 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, USA. His research interests include mobile security and privacy, IoT security, and applied cryptography.

This paper is a revised and expanded version of a paper entitled 'Experimenting emulator detection schemes for protecting mobile apps against reverse engineering' presented at 2015 International Conference on Platform Technology and Service (PlatCon-15), Jeju, Korea, 26 January 2015.

1 Introduction

Android application installation files – that is, APK files – can be easily tampered with by decompiling and debugging them using Android-based reverse-engineering tools such as apktool (https://code.google.com/p/android-apktool/). Using these reverse engineering tools, malicious users can recover the application's source code and insert malicious code. Then, the application is repacked, and the new (tampered) APK file is distributed (Jung et al., 2013).

To prevent such repackaging attacks by easily restoring the original source code, techniques for preventing static analysis are used (Strazzere, 2012). These techniques consist of schemes that make it difficult to translate the APK file into Smali or Java source code. However, applications using these techniques for preventing static analysis are easily immobilised, and they are open to attack using dynamic analysis tools. Moreover, because dynamic analysis tools generally run on an emulator, they can be neutralised when applications detect that they are running in an emulated environment. In this paper, we analyse the theory behind static analysis schemes and emulator-detection schemes, and we investigate the practicality of these schemes by conducting experiments especially in emulators based on the QEMU emulator (Petsas et al., 2014).

This paper is organised as follows. In Section 2, anti-static-analysis schemes are described. We introduce emulator-detection schemes in Section 3. Section 4 describes the results of the experiments. Finally, in Section 5, we conclude the paper.

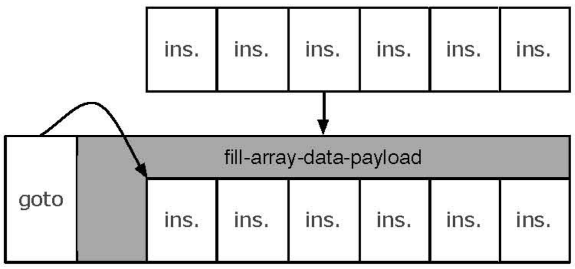
2 Anti-static-analysis schemes

Atypical reverse-engineering tool for Android applications searches for the DEX file among the resources in the APK file, and it then translates this file into Smali code – that is, Android assembly code (https://code.google.com/p/smali/). Generally, malicious users use reverse-engineering tools to acquire the Smali code and insert malicious code into it. This is easy to do because Smali code contains a considerable amount of information, and because this information is readable (Enck et al., 2009). Hence, applications must be protected from repackaging attacks carried out by malicious users. One way to offer protection is by applying anti-static-analysis schemes.

2.1 Fill-array-data-payload

One scheme for protecting applications from attack uses the pseudo-instruction ‘fill-array-data-payload’ to conceal the application’s instructions. The Android operating system uses this instruction when the assembler compiles the Java source into byte code. A pseudo-instruction is an instruction used to make code easier to understand (https://source.android.com/devices/tech/dalvik/instruction-formats.html).

Figure 1 Applying the fill-array-data-payload instruction



To ensure that the pseudo-instruction is not carried out when the application runs, this scheme uses a branching statement, etc., to modify the byte code such that the pseudo-instruction is not executed. Thus, instead of executing the pseudo-instruction, the subsequent instructions are carried out. As a result, this technique does not interfere with the application’s execution, and the decompiler mistranslates the byte code as a pseudo-instruction and outputs incorrect Smali code (Schulz, 2012).

This scheme inserts pseudo-instructions at the beginning of any class method that must be protected from malicious analysis, as shown in Figure 1. Then a branching statement is inserted in order to ensure that the pseudo-instruction is not executed. Consequently, there is no effect while the application runs. Yet, when parsing the DEX file, a decompiler will wrongly perceive the instruction from the original method as simple data entered into the array, and it will display 16 meaningless antilogarithmic values.

2.2 Method hiding

The method-hiding scheme analyses the DEX file format within an APK file in order to manipulate the byte that represents the method information. This way, the decompiler or disassembler, when parsing the DEX file, will read the method as faulty. Thus, one can hide the original method by generating an error resulting from the tools' mistranslation of the method information (Apvrille, 2013).

Figure 2 Original state of the class methods

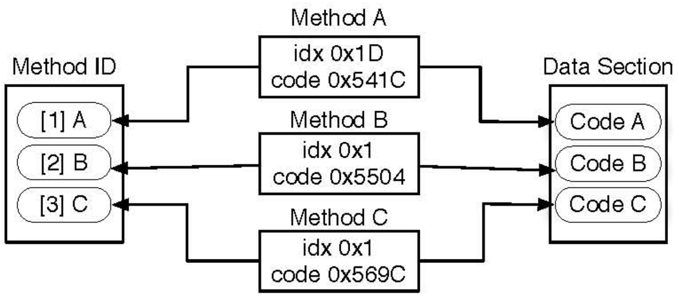
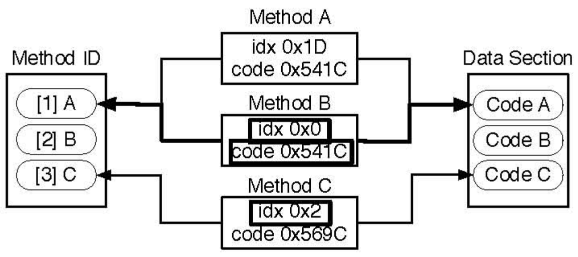


Figure 2 shows the information represented by each function within the classes in the `class_def_table` of the DEX file (https://source.android.com/devices/tech/dalvik/dex-format.html). The `Class_data_item` includes all classes, and their function and field are defined in their designated locations.

Figure 3 shows a modified version with which Method B is hidden. The process for hiding the method proceeds as follows. First, the method idx value for Method B is modified to 0, such that it leads to MethodAin the method list. In addition, in order for the `code_off` value of Method B to lead to the code for Method A, the `code_off` value is modified to that of Method A. Then, to ensure that the decompiler finds Method C, the `method_idx` of Method C is modified to the sum of the `method_idx` of the function that will be hidden and the original idx of its subsequent function. The reason for this is that if the `method_idx` is not the first element, there will be a conspicuous difference between the previous function and the index.

To run the hidden function, the DEX file for the application is opened and restored to its pre-modification state. Once the function hidden by the modified DEX file is found, it can be executed. To run the DEX file for a running application, the `openNonAsset` function in the `android.content.res.AssetManager` can be used. Because the `openNonAsset` function cannot be directly accessed, a Java reflection is also used. Using the `openNonAsset` function, the DEX file is stored in the array. This stored DEX file is then configured to the pre-modification format of the DEX file. Thus, the method that concealed the function is reverted back to its original state. The reset DEX array is opened as a new class using the `openDexFile` function. Then, the original class is loaded using the `defineClass` function. Finally, the `getDeclaredMethods` function brings the list of functions in the class. Once the hidden function is found, it is called. When the method-hiding technique is applied and a decompiler is used, the Smali code for the hidden function is not printed out.

Figure 3 Modified state of the class methods



2.3 Anti-static-analysis scheme using the Zip format

The general structure for a Zip file format comprises N file entries and N central directories (https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT). Modifying of the universal bit of the central directory's file header can prevent static analysis. The first bit of the universal bit denotes the encryption status. If this bit is set to 1, the decompression program will consider the file encrypted, even if it is not. Consequently, the decompiler is unable to decompress the file.

In addition, among the information included in file entry and central directory, there is a field that indicates compressed information. The information regarding the compression method is labelled 'Compression.' Likewise, the size of the compression file is labelled 'CompressedSize.' and the size of the original file as 'UncompressedSize.'

Whether a file is compressed can be determined by comparing the value in the CompressedSize field with that of the UncompressedSize field. If the values of these two fields are the same, the file has not been compressed. If the two fields' values are different, it is a compressed file.

Based on the information above, if the Compression field value is falsified for a central directory that has identical CompressedSize and UncompressedSize values, the decompiler cannot decompress the file. Thus, when this Zip-based method is applied, the file is protected from reverse engineering.

3 Emulator-detection schemes

3.1 Using device information

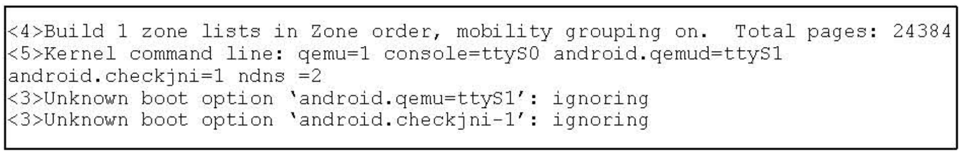
Using the information extracted from real devices and emulators, real devices can be distinguished from emulators (http://developer.android.com/reference/packages.html). For example, an emulator can be discerned using the electronic serial number – i.e., the International Mobile Equipment Identity (IMEI). Real devices have a unique IMEI value, whereas emulators have a uniform number such as `000000000000000`.

Device information other than the IMEI can also be used to differentiate real devices from emulators, including the line number, model name, and brand.

3.2 Using kernel messages

Figure 4 is a screenshot that shows how the Android's Kernel messages can be extracted using an ADB shell (http://linux.die.net/man/3/klogctl). When the system is running in a QEMU emulator environment, information regarding the QEMU environment will be displayed in the Kernel message, as seen in Figure 4 (http://wiki.qemu.org/Main_Page). As a result, using kernel messages, one can determine that a system is running in a QEMU environment.

Figure 4 Printed kernel log



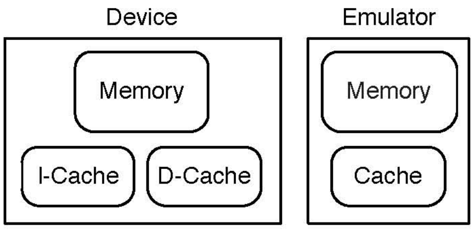
```
<4>Build 1 zone lists in Zone order, mobility grouping on.  Total pages: 24384
<5>Kernel command line: qemu=1 console=ttyS0 android.qemud=ttyS1
android.checkjni=1 ndns =2
<3>Unknown boot option 'android.qemu=ttyS1': ignoring
<3>Unknown boot option 'android.checkjni-1': ignoring
```

3.3 Using the cache architecture

The cache is an intermediary storage device used to reduce the difference in speed between the CPU and memory. It operates by storing a part of the memory area used by the CPU, and then storing altered data back into the memory. By investigating how the cache is operating, an emulated environment can be detected (Bramley, 2010).

Figure 5 shows the cache of a real device and that of an emulator. The real device has two the I-caches storing instructions and a D-cache storing data. The emulator, by contrast, stores both the instructions and data in a single cache (Schulz, 2013).

Figure 5 Cache architecture for a device and an emulator



When the CPU looks for the data, it searches the data in the cache, because the cache is closer than RAM and faster. If the data which is searched by the CPU exists in the cache, this situation is known as a cash HIT. In contrast, when the cache does not have the data, it is called a cache MISS. When a cache MISS is occurred, the data in the RAM is copied into the cache (https://en.wikipedia.org/wiki/Cache).

Owing to the difference between the structure of their respective caches, emulators and real devices can be differentiated. Because real devices have both I- and D-caches, a cache-coherency problem can occur, resulting in the execution of random code. Emulators use only a single cache, and as such they do not encounter problems with cache coherency and all code is executed sequentially.

Figures 6 and 7 show the results from alternately writing Code A and Code B to a specific memory address and then executing this code (Sloss et al., 2004).

Figure 6 Cache process for a real device

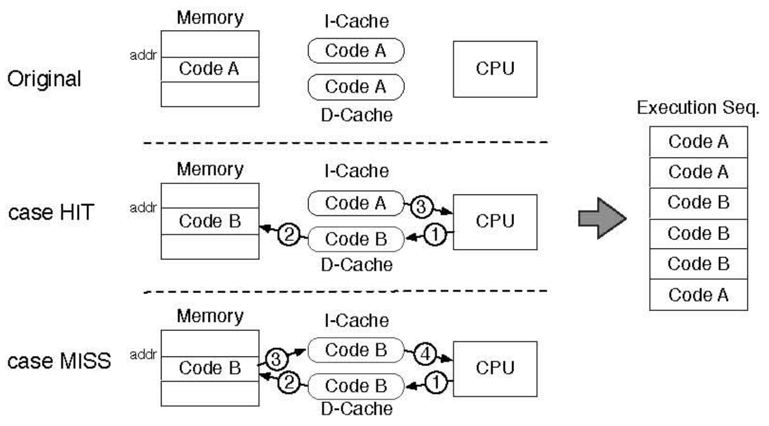


Figure 7 Cache process of an emulator

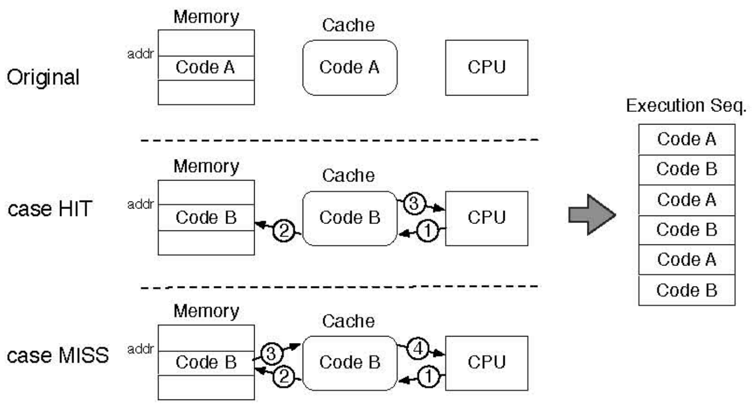


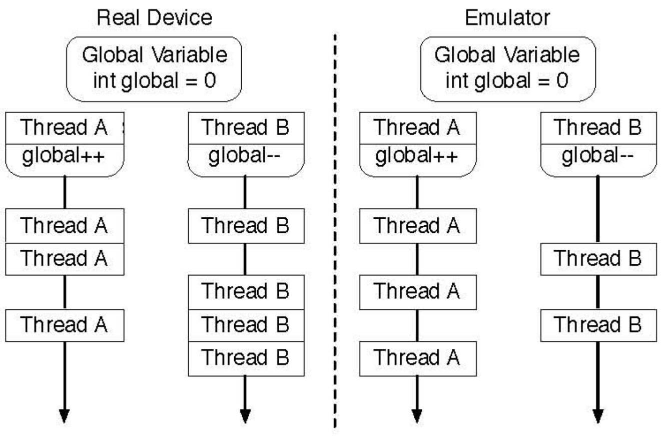
Figure 6 shows the process by which the cache operates in a real device. In the original process, Code B is written into the memory, and the code in that specific memory address is executed. This can result in either a HIT or MISS. If a HIT occurs, the D-cache and the memory are updated with Code B. Because the memory address is already stored in the I-cache, however, the I-cache is not updated. Thus, because the I-cache remains unchanged, Code A is executed. In the case of a MISS, on the other hand, there is no specific memory address pertaining to the I-cache. This causes the I-cache to be updated with Code B, which is then executed. Because of this process, real devices execute code in random order.

Figure 7 shows the cache process for an emulated device. As in Figure 6, Figure 7 shows that in the original state, Code B is written to a specific memory address and executed. When a HIT occurs, Code B is written in the cache and memory. Accordingly, Code B is executed because it is within the cache. In the case of a MISS, the process is the same as the one in Figure 6. Consequently, the code that the emulator intends to execute is in fact executed, and this code is always executed sequentially.

3.4 Using thread synchronisation

A thread is a model that allows for the use of multiple execution flows within a single program. The thread-scheduling method in an emulator operates differently from when it is used in a real device (Matenaar and Schulz, 2012).

Figure 8 Identifying an emulator from regular thread scheduling



As shown in Figure 8, because of the difference between a real device and emulator in terms of the thread-scheduling algorithm, the final output value when a thread is executed in a real device differs when it is executed in an emulator. In a real device, a race condition occurs, whereas this condition does not occur in an emulator (http://en.wikipedia.org/wiki/Race_condition).

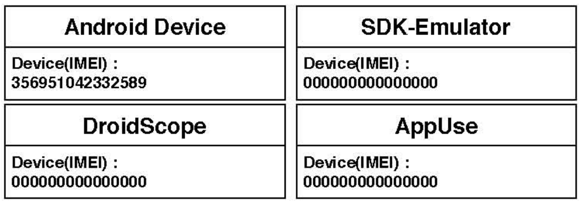
4 Experimental results

All experiments were conducted using an application developed with Android API level 15. The application contained some native code, and it was implemented on three kinds of emulators and one real device. Notably, all of the schemes tested can be applied to any application in order to differentiate emulators from real devices.

4.1 Identifying unique device information

An emulator can be detected using previously secured device-identification information. Figure 9 shows the test results for distinguishing between the real device and the emulators by verifying the IMEI value. For the real Android device, an IMEI value that is unique to the device was found, whereas the SDK-emulator (http://developer.android.com/sdk/index.html; https://code.google.com/p/decaf-platform/; Yan and Yin, 2012; https://appsec-labs.com/appuse/) emulator tools each produced NULL values.

Figure 9 Results from checking the IMEI values

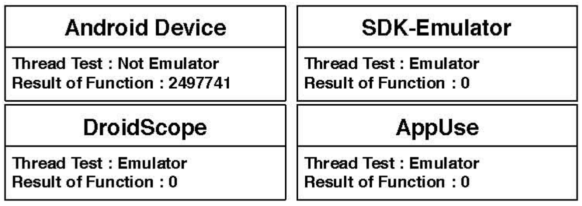


Android Device	SDK-Emulator
Device(IMEI) : 356951042332589	Device(IMEI) : 000000000000000
DroidScope	**AppUse**
Device(IMEI) : 000000000000000	Device(IMEI) : 000000000000000

4.2 *Identifying different execution sequences*

The method for detecting emulators using the thread-scheduling test used two threads. With one thread, the values of global variables initialised as 0 increased in increments of 1. With the other thread, these values decreased by increments of one. Therefore, if the value of the final execution is 0, we know that the application is being run on an emulator, because the two threads were scheduled alternately, as shown in Figure 10. This is because scheduling is performed randomly in the real device, meaning that the final value will not be uniform.

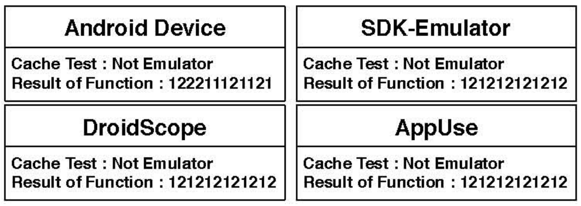
Figure 10 Results of the thread-scheduling test



Android Device	SDK-Emulator
Thread Test : Not Emulator Result of Function : 2497741	Thread Test : Emulator Result of Function : 0
DroidScope	**AppUse**
Thread Test : Emulator Result of Function : 0	Thread Test : Emulator Result of Function : 0

The detection method using the difference in cache structure between two different codes, A and B, records the number 1 for the integer array of code A. When executing the same sequence for Code B, the method records the number 2. Figure 11 shows that, because emulators have only a single cache, they can be identified when the numbers 1 and 2 are stored as a sequence.

Figure 11 Results from the cache test



Android Device	SDK-Emulator
Cache Test : Not Emulator Result of Function : 122211121121	Cache Test : Not Emulator Result of Function : 121212121212
DroidScope	**AppUse**
Cache Test : Not Emulator Result of Function : 121212121212	Cache Test : Not Emulator Result of Function : 121212121212

5 Conclusions

It is not difficult to make Android mobile applications reversible. They can be redistributed after modifying some parts of the code and repackaging the application. When the application is analysed using dynamic analysis tools, its structure and critical functions are revealed. This vulnerability threatens the ecosystem of Android applications. Protection schemes, therefore, are essential to avoid analysis and repackaging attacks. In addition, protecting applications is important in order to defend intellectual property rights from unspecified malicious users.

Among these protection schemes are techniques that check the execution environment. Most dynamic analysis tools run on an emulator. As such, when by detecting emulators, we can protect the application from analysis.

In this paper, we implemented actual schemes designed to counter dynamic analysis environments running Android applications, and we reported the results. In particular, these schemes used the device information, kernel messages, cache architecture, and thread synchronisation to prevent the malicious use of analysis tools. Henceforth, using these results as a basis, we hope to propose an even more effective method for preventing reverse engineering.

Acknowledgements

This research was supported by Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF-2014K1A1A2043029).

References

Android API [online] http://developer.android.com/reference/packages.html (accessed 14 October 2015).

Android SDK [online] http://developer.android.com/sdk/index.html (accessed 14 October 2015).

Apktool [online] https://code.google.com/p/android-apktool/ (accessed 14 October 2015).

AppUse [online] https://appsec-labs.com/appuse/ (accessed 14 October 2015).

Apvrille, A. (2013) 'Playing hide and seek with Dalvik executables', *hacktivity*.

Bramley, J. (2010) 'Caches and self-modifying code', *ARM Connected Community* [online] http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-selfmodifying-code (accessed 14 October 2015).

CPU Cache [online] https://en.wikipedia.org/wiki/Cache (accessed 14 October 2015).

Dalvik Executable Format [online] https://source.android.com/devices/tech/dalvik/dexformat.html (accessed 14 October 2015).

DroidScope [online] https://code.google.com/p/decaf-platform/ (accessed 14 October 2015).

Enck, W., Ongtang, M. and McDaniel, P. (2009) 'Understanding android security', *IEEE Security & Privacy*, Vol. 7, No. 1, pp.50–57.

Instruction Formats [online] https://source.android.com/devices/tech/dalvik/instructionformats.html (accessed 14 October 2015).

Jung, J.H., Kim, J.H., Lee, H.C. and Yi, J.H. (2013) 'Repackaging attack on android banking applications and its countermeasures', *Journal of Wireless Personal Communications*, Vol. 73, No. 4, pp.1421–1437.

Kernel Message [online] http://linux.die.net/man/3/klogctl (accessed 14 October 2015).

Matenaar, F. and Schulz, P. (2012) 'Detecting Android Sandboxes', *Dexlabs* [online] http://www.dexlabs.org/blog/btdetect (accessed 14 October 2015).

Petsas, T., Voyatzis, G. and Athanasopoulos, E. (2014) 'Rage against the virtual machine: hindering dynamic analysis of Android malware', *ACM European Workshop on System Security*.

QEMU [online] http://wiki.qemu.org/Main_Page (accessed 14 October 2015).

Race Condition [online] http://en.wikipedia.org/wiki/Race_condition (accessed 14 October 2015).

Schulz, P. (2012) 'Dalvik bytecode obfuscation on Android', *Dexlabs* [online] http://www.dexlabs.org/blog/bytecode-obfuscation (accessed 14 October 2015).

Schulz, P. (2013) 'Android emulator detection by observing low-level caching behavior', *Bluebox* [online] https://bluebox.com/blog/technical/android-emulator-detection-byobserving-low-level-caching-behavior/ (accessed 14 October 2015).

Sloss, A.N., Symes, D., Wright, C. and Athanasopoulos, E. (2004) 'Caches, writing and optimizing ARM assembly code', *ARM System Developer Â¡Â¯s Guide*, 1st ed.

Smali [online] https://code.google.com/p/smali/ (accessed 14 October 2015).

Strazzere, T. (2012) 'Dex education: practicing safe Dex', *BlackHat USA*.

Yan, L.K. and Yin, H. (2012) 'DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis', *USENIX Security*.

Zip File Format [online] https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT (accessed 14 October 2015).