Chapter 14

FORENSIC-READY SECURE iOS APPS FOR JAILBROKEN iPHONES

Jayaprakash Govindaraj, Rashmi Mata, Robin Verma and Gaurav Gupta

Abstract Apple iOS is one of the most popular smartphone operating systems, but it restricts the installation of apps that are not from the Apple App Store. As a result, users often jailbreak their iPhones to defeat this restriction. Jailbroken iPhones are making their way into enterprises that have a Bring Your Own Device (BYOD) policy, but these devices are often barred or restricted by mobile device management software because they pose security risks. This chapter describes the iSecureRing solution that secures mobile apps and preserves the dates and timestamps of events in order to support forensic examinations of jailbroken iPhones. An analysis of the literature reveals that iSecureRing is the first forensic-ready mobile app security solution for iOS applications that execute in unsecured enterprise environments.

Keywords: Jailbroken iPhones, enterprise environments, forensic examinations

1. Introduction

According to a 2013 Pew report [19], 40.98 % of the smartphones used by adult Americans are Apple iPhones. Apple's iOS operating system does not allow the installation of applications, extensions and themes that are not obtained from the Apple App Store. As a result, users frequently jailbreak their devices to obtain root access and defeat the installation restrictions [4]. A jailbroken iPhone allows the retrieval of applications and their associated data, potentially compromising the security of the applications and the confidentiality of the data [4, 16].

Since jailbreaking is a reality [8, 11], it is increasingly important to design mobile applications that can run securely on jailbroken iPhones. The requirement of having an application execute securely in an unsecure environment is critical to scenarios where a proprietary applica-

tion should work without impacting enterprise security. At this time, enterprises that have a Bring Your Own Device (BYOD) policy generally detect and restrict jailbroken iPhones using mobile device management software such as Citrix's XenMobile and IBM's Endpoint Manager. Thus, employees have to un-jailbreak their iPhones or install enterprise applications on other approved devices. The solution proposed in this chapter enables enterprises to install their applications securely on jailbroken iPhones. New apps and existing apps can be secured and be made forensic-ready. The forensic readiness of the apps enables enterprises to check if the apps run securely and also ensures that forensic artifacts are available in the event of security incidents.

2. Related Work

D'Orazio et al. [2] have proposed a concealment technique that enhances the security of unprotected (class D) data that is at rest in iOS devices, along with a deletion technique to reinforce data deletion in iOS devices. Hackers and malicious users resort to techniques such as jailbreaking, running an app in the debug mode, reverse engineering, dynamic hooking or tampering in order to access or compromise sensitive data stored by iOS apps:

- **Jailbreaking:** Attackers use jailbreaking to obtain system-level (root) access to iOS devices, potentially compromising the security of applications and their associated data [15].
- **Debugger Mode:** Attackers run targeted applications in the debug mode, obtain memory dumps and overwrite the memory with malicious code [9, 13].
- **Reverse Engineering:** Apps from the Apple Store are encrypted using Apple's Fairplay DRM, which complicates the task of reverse engineering binaries. However, an attacker can overwrite the encryption information of an application in a jailbroken device to obtain the memory dump and analyze it to create new attacks [3, 16].
- **Dynamic Code Hooking:** After a device is jailbroken, an attacker can hook malicious code to an app at runtime in order to bypass security checks, potentially compromising the security of the application and its data [20].
- **Tampering:** Attackers can modify the dates and timestamps of artifacts in order to cover their tracks. Verma et al. [21] have recently proposed a mechanism for preserving dates and timestamps in support of forensic examinations of Android smartphones.

This chapter presents a technique for protecting applications and data in jailbroken iOS devices. In the event of a security incident, the technique can be used to support a forensic examination of a jailbroken device.

3. Implementation Methodology

The solution has two modules: (i) a static library that wraps apps running on jailbroken devices with an extra layer of protection, making them difficult to crack and preventing access to their data; and (ii) a module that preserves authentic dates and timestamps of events related to the secured apps to support forensic examinations. The captured dates and timestamps are stored outside the device on a secure server or in the cloud. The modules are discussed in following subsections.

3.1 Securing Apps

The static library, which is designed to secure apps, incorporates APIs that may be used to identify and mitigate security vulnerabilities in jailbroken iPhones [6]. Functions in the library include: `isCheck1()`, which checks if an iPhone is jailbroken; `isCheck2()`, which checks if an application is running in the debug mode; `enableDB()`, which disables the `gdb` (debugger) for a particular application (process); `isAppC()`, which checks if an application binary is encrypted and also checks the integrity of application bundle files (`Info.Plist`); `initialize()`, which checks if static library functions are hooked; `CheckA()`, which checks if critical methods (functions) passed as arguments are hooked; `CheckS()`, which checks if methods (functions) related to SSL certificate validation are hooked; `createCheck()` and `createCheckTest()`, which check if an application has been tampered with; and `resetZeroAll()`, which wipes sensitive data from memory.

3.2 Preserving Dates and Timestamps

The dynamic library has been created using the MobileSubstrate framework. This framework provides APIs for adding runtime patches or hooks to system functions in jailbroken iOS devices [18]. The solution architecture shown in Figure 1 incorporates three components:

- **Dynamic Library (dylib):** This component hooks system open calls and captures kernel-level dates and timestamps of selected files and writes them to the log file. It is loaded into running applications. Filters are applied so that it is only loaded into specified applications.

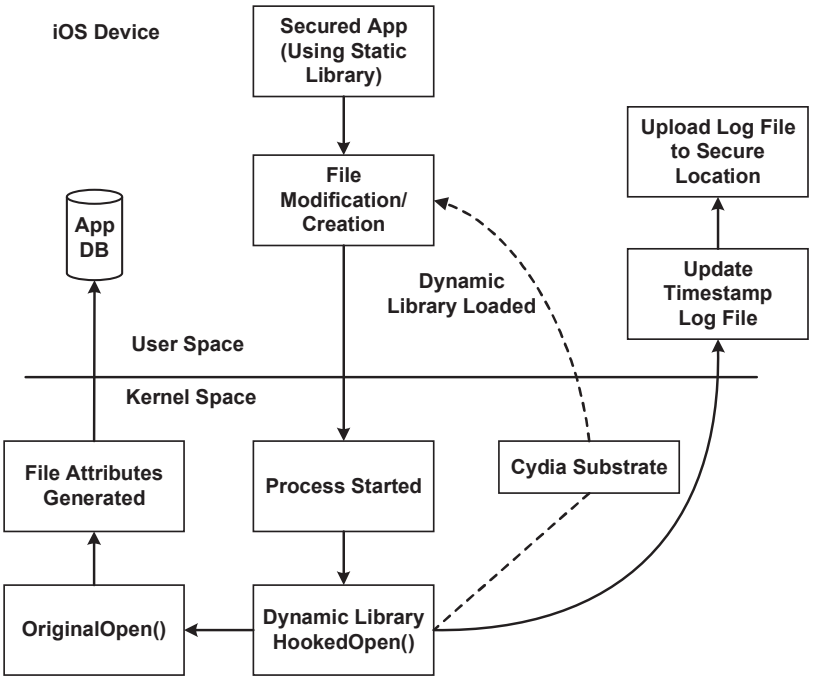

Figure 1. Solution architecture for preserving date and timestamps.

- **Timestamp Log File:** This component is stored in the internal memory of an iPhone. It is not directly accessible to applications, which secures it from unauthorized deletion.
- **Log File:** This component is generated by the DLL. It is uploaded at regular user-defined intervals to an external server or cloud storage based on network connectivity.

3.3 Static Library

The static library is designed to secure applications and their associated data. The library wraps apps in an additional layer of protection, which makes them more difficult to crack in a jailbroken iOS device. The static library contains several APIs (Table 1) that can be used to identify security vulnerabilities in jailbroken devices. The library implements the detection of jailbroken devices, the disabling of application debuggers, the checking of application encryption (for App Store binaries) and the detection of dynamic code hooking. Note that the function names are intentionally not very descriptive in order to enhance code obfuscation and hinder malicious reverse engineering efforts.

Table 1. Static library APIs.

API	Description
`isCheck1()`	Checks if a device has been jailbroken
`isAppC()`	Checks if the application encryption provided by the App Store is intact
`enableDB()`	Disables the application debugger
`isCheck2()`	Checks if an app is running in the debug mode
`Initialize()`	Checks if library APIs are hooked by method swizzling techniques
`checkA()`	Checks if a function is hooked by a method swizzling technique
`checkS()`	Checks if the SSL validation methods provided by the iOS SDK are hooked
`makeZero()`	Finds the data portion of object memory and zeroes it out
`encPwd()`	Encrypts object data in memory using a secret
`decPwd()`	Decrypts object data in memory using a secret
`listed()`	Adds an object to the pointer list used by the APIs
`unlisted()`	Removes an object from the pointer list
`resetAllZero()`	Wipes all tracked objects
`createCheck()`	Provides and statically stores a string of all the tracked memory addresses and object checksums
`createCheckTest()`	Checks if the current memory states of all the tracked objects match their states when `checksumMem()` was called

3.4 Dynamic Library

The dynamic library was created using the MobileSubstrate framework, now known as the Cydia Substrate [18]. The framework provides a platform and APIs for adding runtime patches or hooks to system functions as well as other applications on jailbroken iOS and rooted Android devices. The MobileSubstrate framework incorporates three components: (i) Mobilehooker; (ii) Mobileloader; and (iii) Safemode.

- **Mobilehooker:** This component replaces the original function with the hooked function. Two APIs may be used for iOS devices: (i) `MSHookMessage()`, which is mainly used to replace Objective-C methods at runtime; and (ii) `MSHookFunction()`, which is used to replace system functions, mainly native code written in C, C++ or assembly.

- **Mobileloader:** Cydia Substrate code is compiled to create the dynamic library, which is placed in the directory */Library/Mobile Substrate/DynamicLibraries/* in jailbroken iOS devices. The main task of Mobileloader is to load the dynamic library into running

applications. The Mobileloader initially loads itself and then invokes `dlopen` on all the dynamic libraries in the directory and loads them at runtime.

The dynamic libraries are configured using Property List (PList) files, which act as filters, controlling if a library should be loaded or not. The PList file should have the same name as that of `dylib` and should be stored in the same directory as `dylib`. The PList should contain a set of arrays in a dictionary with the key Filter. The other keys used are: (i) Bundles (array) – the Bundle ID of a running application is matched against the list, if a match occurs, then `dylib` is loaded; (ii) Classes (array) – the `dylib` is loaded if one of the specified Objective-C classes in the list is implemented in the running application; and (iii) Executables (array) – `dylib` is loaded if an executable name in the list matches the executable name of the running application.

An example is:

```
Filter = Executables = ("mediaserverd"); Bundles =
   ("com.apple.MobileSlideShow"); Mode = "Any";;
```

In the example, the filter ensures that `dylib` is loaded only for the iOS built-in application Photos, whose executable name matches `mediaserverd` or Bundle ID is `com.apple.MobileSlideShow`. The Mode key is used when there are more than one filters. By specifying Mode = Any, `dylib` is loaded if one of the filters has a match.

- **Safemode:** In this mode, all third-party tweaks and extensions are disabled, preventing the iOS device from entering the crash mode. Following this, the broken `dylib` can be uninstalled from the device.

Compilation Procedure. The Theos [10] development suite was used to edit, compile and install the dynamic library on a device. It provides a component named Logos, which is a built-in pre-processor-based library designed to simplify the development of MobileSubstrate extensions. In order to compile the dynamic library, Theos must be installed on a Mac machine. A Mac OS X has most of the tools required by Theos; however, Xcode command line tools must be installed if they are not present. Additionally, it is necessary to install the `ldid` tool, which is used to sign apps or tweaks so that they can be installed on jailbroken iOS devices.

To start the project, it is necessary to obtain all the iOS private headers of the functions intended to be hooked. The headers can be dumped

using the Class-Dump-Z command line tool [5]. This reverse engineering tool provides complete header information of the Objective-C code of an iOS application. Dumping the headers can take some time because headers from all the frameworks, including private frameworks, are also collected. The dumped headers are saved in a folder with the corresponding framework name. Instead of dumping the headers, headers collected by other researchers can be used (e.g., headers from GitHub). All the headers are saved at `/opt/theos/include`.

The next step is to create the Theos project. This involves executing the file `/opt/theos/bin/nic.pl` from the command line and choosing the project template, name, etc. The project type should be library because the goal is to hook a system function. After the project has been created, a new file named `tweak.xm` is found in the project directory; this file is used to store the hooking code.

The following pseudocode for hooking an `open()` system call is added in the `tweak.xm` file:

```
extern "C"
{ int orig\_open(const char *path, int oflags);}
int hijacked\_open(const char *path, int oflags)
{    // do something, then
return orig\_open(path, oflags);
}
\%ctor{ NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
MSHookFunction(open(), \&hijacked\_open, \& orig\_open);
[pool drain];}
```

The `MSHookFunction()` API is used to hook the `open()` system call. The replacement function is `hijacked_open()`. The `makefile` is then modified to add the required frameworks. Note that the Foundation framework is used to create the hooking code. The target SDK version and the architecture needed to support it are also added:

```
TARGET := iPhone:7.0
ARCHS := armv7 arm64
ProjectName\_FRAMEWORKS = Foundation

Once done, call make from command line as below.
xyz:test xyz make
Making all for application test...
Copying resource directories into the application wrapper...

Signing test...
```

The project is then compiled and a DLL is created in the `obj` folder.

DLL Loading. After the DLL is created, it can be installed on a device by the Theos suite using the command `make package install`. This command creates a Debian package of the DLL and installs it in the proper location on the device. Before this is done, the environment variable must be set to `export THEOS_DEVICE_IP=iPhone Device IP`. Next, the package is transferred to the device for installation via SFTP. The iOS device should be on the same network as the computer used for development [20].

4. Preventing Attacks and Anti-Forensics

The section discusses how attacks and anti-forensic approaches can be mitigated using the static and dynamic libraries.

4.1 Using the Static Library

- **BOOL isCheck1():** This function is used to check if an iOS device is jailbroken. It returns yes if the iOS device is jailbroken; otherwise no. This function can be called before application launch.

- **API for Checking Debug Mode:** The application exits when launched in the debug mode using the `enableDB()` function. This function can be called from `main()` and from elsewhere in the project to disable debugging at any stage. By calling `enabledDB()` in `main()` or before app launch, the application can be prevented from running in the debug mode. Therefore, the function should be called in the release mode.

- **isCheck2():** This function gives information about how the application is running. If the application was started in the debug mode, then a value of one is returned; otherwise zero is returned.

- **BOOL isAppC(char* inBundlePath):** This function checks if the application has been hacked. The parameter `inBundlePath` can be any character pointer; it is only added for obfuscation and is not used inside the function. It includes an app encryption check (if the App Store encryption is broken), signer identity checks, etc. If the app is cracked, the function returns yes; otherwise no. The function is primarily used to check if App Store binaries are cracked.

- **int Initialize():** This function checks if the APIs in the static library are themselves hooked. The function has to be called initially, preferably during app launch, to check if the library APIs

are hooked by method swizzling, after which the appropriate actions must be taken. If the functions are hooked, then it makes no sense to use the APIs to protect applications.

- **int checkA(const char* MCl, const char* MFr, const char* MFn, void *funcPTR):** This function checks if any hooking is done for a critical method within an application passed as an argument to the function. The function returns one if no hooking is discovered and zero if a function is hooked. It requires the method name, method class and the path of the framework (for a framework method) or app bundle path (for an application method).
- **int checkS():** This function checks if SSL certificate validation methods provided by the iOS SDK are hooked. This function is invoked within an application before calling SSL validation methods so that the proper actions can be taken. The function returns one if there is no hooking and zero if a function is hooked.
- **makeZero(obj):** This function is used to zero the value of a sensitive variable after its use.
- **encPwd() and decPwd():** These APIs are used for encrypting sensitive data immediately after the data is created and decrypting the data only during its use. After the sensitive data has been used, it should be cleared from memory permanently.
- **listed() and unlisted():** These functions track several objects in order to clear them from memory simultaneously. Sensitive objects are added to the list to keep track of them; they are all cleared at one time using an API. For example, when a device is locked and/or an app is closed (hidden or terminated), it may be necessary to wipe all the sensitive data. In this case, it is necessary to add `resetZeroAll()` to the state-change notify functions in `AppDelegate`. Several tools are available for attackers to modify the values of critical data and change the behavior of an application at runtime. Such modifications can be tracked using the `createCheck()` and `createCheckTest()` APIs to create a checksum of the critical data and check it periodically to ensure that the data is not modified by an attacker.

4.2 Using the Dynamic Library

Whenever files are modified, accessed or created, the hijacked `open()` call is invoked and the modified, accessed, created dates and timestamps (MAC DTS) are captured and stored in the log file. The log file is stored

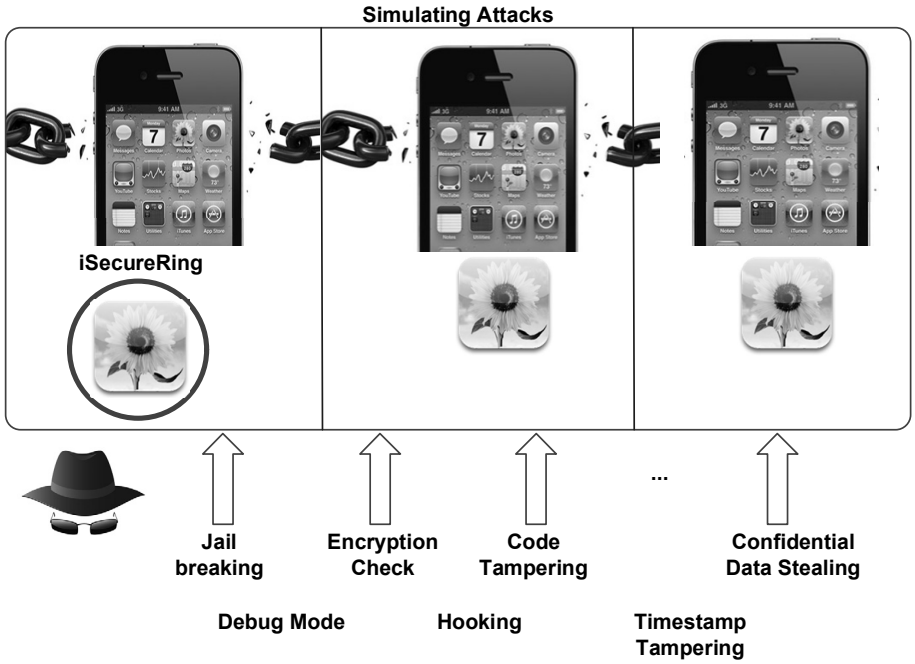

Figure 2. Simulating attacks on devices.

outside the iPhone at a secure location such as a server or in the cloud. The information in the log file can be used in a forensic investigation of the smartphone in the event of a security incident.

5. Experimental Results

The experiments involved the creation of two apps, one without any protection and the other protected by iSecureRing. The apps were then deployed on a jailbroken iPhone 4 (iOS 7.0.6).

A series of attacks were simulated on the apps and their data to validate the proposed solution (Figure 2). At the application level, the apps were subjected to various attacks to exploit the lack of binary protection [2]. The results in Table 2 demonstrate that an app with iSecureRing running on a jailbroken iPhone (Row 3) is just as secure as a normal app running on a non-jailbroken iPhone (Row 1).

Performance benchmarking was conducted for the three cases considered in the experiments. Figure 3 summarizes the results of the initial tests (five runs). The results show no significant differences in device performance.

Table 2. Attacks and results.

iPhone 4 (iOS 7.0.6)	Jail-broken	Debug Mode	Encryption Check	Hooking	Code Tampering
Not Jailbroken (App without protection)	Yes	No	No	No	No
Jailbroken (App without protection)	N/A	Yes	Yes	Yes	Yes
Jailbroken (App with iSecureRing)	N/A	No	No	No	No

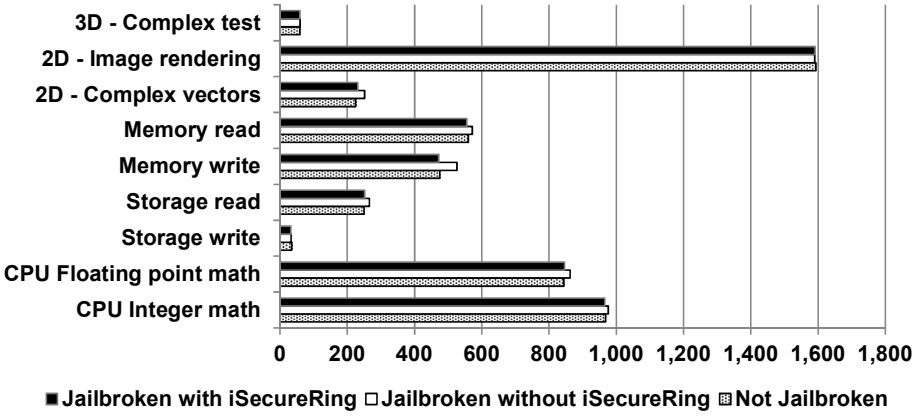

Figure 3. Performance benchmark results.

```
file=/System/Library/PrivateFrameworks/TextInput.framework/Info.plist
Mtime=Wed Feb 19 11:10:02 2014
 Atime=Wed Feb 19 11:10:02 2014
 Ctime=Wed Feb 19 11:10:02 2014
 Flag=0

file=/var/mobile/Media/PhotoData/Thumbnails/V2/DCIM/100APPLE/IMG_0002.PNG/5003.JPG
Mtime=Sun Mar  2 23:27:24 2014
 Atime=Sun Mar  2 23:27:24 2014
 Ctime=Sun Mar  2 23:27:24 2014
 Flag=0
```

Figure 4. MAC DTS logs for an image file.

iSecureRing also helps detect attempts to exploit known or unknown vulnerabilities by capturing the timestamps of activities associated with a secured app. One experiment involved timestamp tampering attempts on images from Apple's Photo app. iSecureRing successfully captured all the events in the log. An analysis of the log clearly revealed the tampering attempts. Figure 4 shows the MAC DTS logs for one of the images.

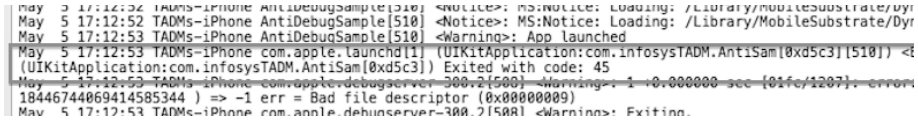

Figure 5. Preventing a debug mode attack.

Debug Mode Attack. Figure 5 shows a screenshot of an Xcode application running in the debug mode [17] while it was being protected by iSecureRing.

Encryption Attack. Clutch [17] was used to simulate an encryption attack (Clutch may be downloaded from `cydia.iphonecake.com`). iSecureRing includes a check to determine if application encryption is intact by analyzing encryption information in the binary and also the `cryptid` flag value. If the application encryption is found to be broken, then the user is alerted and the application behavior can be changed at runtime.

Hooking. A hooking attack was simulated by hooking SSL validation methods. It is possible to launch a man-in-the-middle attack on even HTTPS requests to understand, steal and modify the requested data. iSecureRing incorporates several checks to identify the hooking of critical methods such as SSL validation and authentication. Applications that use iSecureRing are protected against the attacks because the user is alerted and the app behavior can be changed.

Code Tampering. A code tampering attack was simulated using Cycript [7], a JavaScript interpreter. The tool was used to modify iOS application behavior at runtime (e.g., bypassing some authentication checks and accessing critical information from memory). An application compiled with iSecureRing provides APIs for identifying code tampering of critical instance variables and class objects using CRC checksums. Also, APIs are provided for wiping sensitive data from memory.

6. Case Study

The iSecureRing implementation was successfully used to secure an iPhone mobile app for a leading bank in India.

Problem Statement. The bank had a security incident in which the application running on a jailbroken device revealed sensitive data.

Jailbreaking Attack. The attacker had used the latest jailbreaking tool for iOS 7.1.2 Pangu [1]. The attacker also used certain tweaks to de-

feat jailbreak detection. Some tweaks within Cydia bypass the jailbreak detection check of an application and make the application run normally even on jailbroken devices. One such tweak is xCon [12], which hooks low-level APIs used for jailbreak detection such as file-related APIs and other system calls, thus bypassing the jailbreak detection functions used in the application. No configuration is required for xCon, a MobileSubstrate dynamic library that can be installed from Cydia.

iSecureRing Results. The bank app was secured using iSecureRing. The jailbreak detection function was robust enough to check if the device was jailbroken and if any jailbreak detection bypassing functions were present. The solution was tested with xCon 39 beta 7 on iPhone 4 devices with iOS version 7.1.2. xCon was unable to bypass the jailbreak detection techniques used by iSecureRing. The iSecureRing solution checks for the existence of file-related system function hooking, rendering xCon-like tweaks useless. The solution also mitigates the jailbreak detection bypassing mechanism provided by the xCon dynamic library.

7. Conclusions

The iSecureRing solution secures apps on jailbroken iOS devices. The static library helps detect security vulnerabilities and alerts users to take appropriate actions. The dynamic library helps detect malicious tampering of data by storing authentic copies of MAC DTS values on a local server or in the cloud; this also supports offline digital forensic investigations after security incidents. Thus, iSecureRing enables existing and new apps to be secured and made forensic-ready even if the iOS device has been jailbroken. With enterprises implementing BYOD policies and jailbroken devices making their way into enterprises, the iSecureRing solution helps enterprises mitigate the security risks while enabling employees to use one device for official and personal activities.

Future research will focus on analyzing anomalous interactions with secured apps, blocking attacks and raising alerts. Efforts will also be made to create similar solutions for Android and Windows smartphones.

References

[1] J. Benjamin, How to jailbreak iOS 7.1.x with Pangu 1.1 on Windows, *iDownloadBlog*, June 29, 2014.

[2] C. D'Orazio, A. Ariffin and K. Choo, iOS anti-forensics: How can we securely conceal, delete and insert data? *Proceedings of the Forty-Seventh Hawaii International Conference on System Sciences*, pp. 4838–4847, 2014.

[3] D. Ertel, Decrypting iOS apps (www.infointox.net/?p=114), 2013.

[4] S. Esser, Exploiting the iOS kernel, presented at *Black Hat USA*, 2011.

[5] P. Gianchandani, iOS Application Security Part 2 – Getting Class Information of iOS Apps, Infosec Institute, Elmwood Park, Illinois, 2014.

[6] GitHub, Tools for securely clearing and validating iOS application memory, San Francisco, California (github.com/project-imas/memory-security), 2014.

[7] S. Guerrero Selma, Hacking iOS on the run: Using Cycript, presented at the *RSA Conference*, 2014.

[8] A. Hoog and K. Strzempka, *iPhone and iOS Forensics: Investigation, Analysis and Mobile Security for Apple iPhone, iPad and iOS Devices*, Syngress, Waltham, Massachusetts, 2011.

[9] iPhoneDevWiki, debugserver (iphonedevwiki.net/index.php/Debugserver), 2015.

[10] iPhoneDevWiki, Theos (iphonedevwiki.net/index.php/Theos), 2015.

[11] iPhoneHacks, Jailbreaking your iPhone remains legal in US, but it is illegal to jailbreak your iPad and unlock your iPhone under DMCA, October 26, 2012.

[12] iPhone Wiki, xCon (theiphonewiki.com/wiki/XCon), 2014.

[13] C. Miller, Owning the fanboys: Hacking Mac OS X, presented at *Black Hat Japan*, 2008.

[14] C. Miller, Mobile attacks and defense, *IEEE Security and Privacy*, vol. 9(4), pp. 68–70, 2011.

[15] S. Morrissey, *iOS Forensic Analysis for iPhone, iPad and iPod Touch*, Apress, New York, 2010.

[16] M. Renard, Practical iOS apps hacking, *Proceedings of the First International Symposium on Grey-Hat Hacking*, pp. 14–26, 2012.

[17] B. Satish, Penetration Testing for iPhone Applications – Part 5, Infosec Institute, Elmwood Park, Illinois, 2013.

[18] SaurikIT, Cydia Substrate, Isla Vista, California (www.cydiasubstrate.com), 2014.

[19] A. Smith, Smartphone Ownership 2013, Pew Research Center, Washington, DC, June 5, 2013.

[20] B. Trebitowski, Beginning Jailbroken iOS Development – Building and Deployment, Pixegon, Albuquerque, New Mexico (`brandontreb.com/beginning-jailbroken-ios-development-building-and-deployment`), 2011.

[21] R. Verma, J. Govindaraj and G. Gupta, Preserving date and timestamps for incident handling in Android smartphones, in *Advances in Digital Forensics X*, G. Peterson and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 209–225, 2014.