# iOS anti-forensics: How can we securely conceal, delete and insert data?

Christian D'Orazio[1] Aswami Ariffin[1,2] Kim-Kwang Raymond Choo[1]
[1]Information Assurance Research Group, University of South Australia
[2]CyberSecurity Malaysia, Malaysia
christian.dorazio@cjsoftlab.com, aswamifadillah@gmail.com, raymond.choo@unisa.edu.au

## Abstract

*With increasing popularity of smart mobile devices such as iOS devices, security and privacy concerns have emerged as a salient area of inquiry. A relatively under-studied area is anti-mobile forensics to prevent or inhibit forensic investigations. In this paper, we propose a "Concealment" technique to enhance the security of non-protected (Class D) data that is at rest on iOS devices, as well as a "Deletion" technique to reinforce data deletion from iOS devices. We also demonstrate how our "Insertion" technique can be used to insert data into iOS devices surreptitiously that would be hard to pick up in a forensic investigation.*

## 1. Introduction

With the widespread adoption of ubiquitous smart mobile devices (e.g. iOS and Android devices) and their capacity to act as a general purpose computing platform, they are increasingly seen as a potential attack vector for cyber criminal activities.

Given the increase in mobile devices in everyday life, digital forensics is increasingly being used in the courts. The concept central to mobile (and generally, digital) forensics is digital evidence [17].

Mobile forensics is the process of gathering evidence of some type of an incident or crime that has involved mobile devices. In such circumstances, the expectation is that there has been some accumulation or retention of data on the mobile devices which will need to be identified, preserved and analysed [13][14]. This process can be documented and defined, and be used to uncover evidence of a crime. There has also been an increased interest in anti-mobile forensic techniques by government agencies, the private sector and criminals to securely conceal or destroy data. For example, government agencies (especially those working in national security and intelligence) and the private sector would not want data stored on misplaced or stolen mobile devices to be (forensically) recovered by foreign government agencies, competing businesses, and other actors with malicious intents; and criminals would not want incriminating data to be forensically recovered by law enforcement agencies. Commonly used anti-forensics (also known as counter forensics) techniques and methodologies include secure data deletion, overwriting metadata, avoiding detection, and trail obfuscation [1],[10][11].

While there is a wide range of smart mobile devices, three main operating systems dominate the market, namely Apple iOS, Google Android and RIM Blackberry [12]. iOS devices will be the focus in this paper. Using a 4-digit passcode or alphanumeric password in iOS devices does little to protect a user's data on such devices during forensic examinations. Native iOS applications do not encrypt files such as photos, videos, SMS messages, and other documents using passcode derivation. These files are associated with the *non-protection class key (the Class D key commonly cited as Dkey)*, and, therefore, can be decrypted without the knowledge of the passcode or password of a locked device. When using native iOS applications, file data protection is not a viable option to effectively maintain data confidentiality as the data partition and Class D keys are stored in the Effaceable Storage. These keys are just designed to be rapidly erased on demand and render the entire volume cryptographically inaccessible at once.

There are also known problems in using standard file deletion techniques. For example, the content of the journal file can be indexed to search for the metadata associated with the deleted files [4]. On iOS devices, the metadata may contain per-files keys. The technique described by Burghardt and Feldman [4] requires a certain level of technical expertise and is complicated by the fact that transactions in the journal file are rapidly overwritten, which reduces the chances of data discovery. However, data stored on the iOS devices may still be recoverable if such recovery or forensic activities were undertaken soon after the data has been deleted. To provide a higher level of data

978-1-4799-2504-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HICSS.2014.594

protection assurance, one could develop customised applications that associate stored content with data protection classes A, B, or C (see Table 1) via the relevant API calls provided by Apple iOS software development kits (SDKs).

**Contribution 1**: Instead of developing customised applications, we propose a "Concealment" technique that enhances the security of non-protected (Class D) data that is at rest on iOS devices. This technique, designed to render the content unreadable without the corresponding 256-bit secret, is inspired by the one-time pad (designed by Gilbert Vernam in 1917, see US Patent 1310719) concept, which performs a binary XOR operation on a message (in our context, the per-file key) with a key of the same bit-length (in our context, the *C* value).

We also propose a "Deletion" technique to protect the data on iOS devices, as well as to reinforce data deletion from iOS devices. As pointed out by one of the reviewers, the "Deletion" technique may be more useful to some users than a reversible obfuscation (the "Concealment" technique). However, the "Concealment" technique would be of use to data owner who wants an additional security layer to prevent unauthorised access to the content.

Both techniques produce results on iOS devices that are tethered jailbroken[1], which are persistent and undetected when the device is rebooted and recovers its non-jailbroken state. Neither the iOS device nor the operating system is functionally degraded or compromised.

**Contribution 2**: Our "Insertion" technique can be used to insert data into the devices surreptitiously that would be hard to pick up in a forensic investigation.

## 2. Our proposed iOS anti-forensic technique: Using iPhones as case studies

In order to keep information secure on iOS devices, Apple designed a complex and hierarchical model integrated by different security layers. Generally, data blocks in the user partition are secured with different encryption keys. The data partition file system key, commonly referred as the EMF key, protects special files (catalog file, attributes file, journal file, etc.), metadata, and the file system structure. The special files are required to access the file system payload integrated by folders, user files, and attributes. Every file on the data partition is encrypted with a unique 256-bit key (also known as per-file keys) by the dedicated AES cryptographic engine using AES CBC mode [3]. Each per-file key is wrapped with the key associated with the data protection class (see Table 1) to which the file belongs and stored in the file's metadata.

[1] A tethered jailbreak means that the device will need to be connected to the desktop in order to boot back into a jailbroken state [9]. Tethered jailbreaking is temporary and suitable for digital forensic practitioners as no permanent modifications are made to the OS or user data partition. On the contrary, an untethered jailbreak means that the jailbroken state is permanent.

**Table 1. Data protection classes**

| ID | Name |
|---|---|
| A | NSFileProtectionComplete |
| B | NSFileProtectionCompleteUnlessOpen |
| C | NSFileProtectionCompleteUntilFirstUserAuthentication |
| D | NSFileProtectionNone (No protection) |

Figure 1 depicts the process and cryptographic elements (including data protection classes) that are relevant to our study and necessary to decrypt file contents regardless whether the passcode has been activated (or not). Figure 1 also illustrates how and where the file decryption is affected by our concealment and deletion procedures.

### ➢ Experiment set-up

In our case studies, we used two models of iPhone, namely an iPhone 3GS (storage capacity of 8GBytes) and an iPhone 4 (storage capacity of 16GBytes). Although the phones we used were not the latest versions at the time of the research, the latest iOS version (6.1.2) at the time was installed on both devices. Due to space constraints, we will only present our findings from the iPhone 4 case study (and similar findings were reported in our iPhone 3GS case study).

The iPhone was first reset to factory settings and all previously stored contents were erased to ensure there was no remnant data. We took five pictures (IMG_0001.JPG, IMG_0002.JPG, IMG_0003.JPG, IMG_0004.JPG and IMG_0005.JPG), three videos (IMG_0006.MOV, IMG_0007.MOV and IMG_0008.MOV) using the iPhone's camera, and four screenshots (IMG_0009.PNG, IMG_0010.PNG, IMG_0011.PNG and IMG_0012.PNG) pressing the Home and Sleep buttons. We then undertook the "Concealment" and "Deletion" procedures using the toolkit detailed in Table 2.

Our proposed technique is outlined in Figure 2.

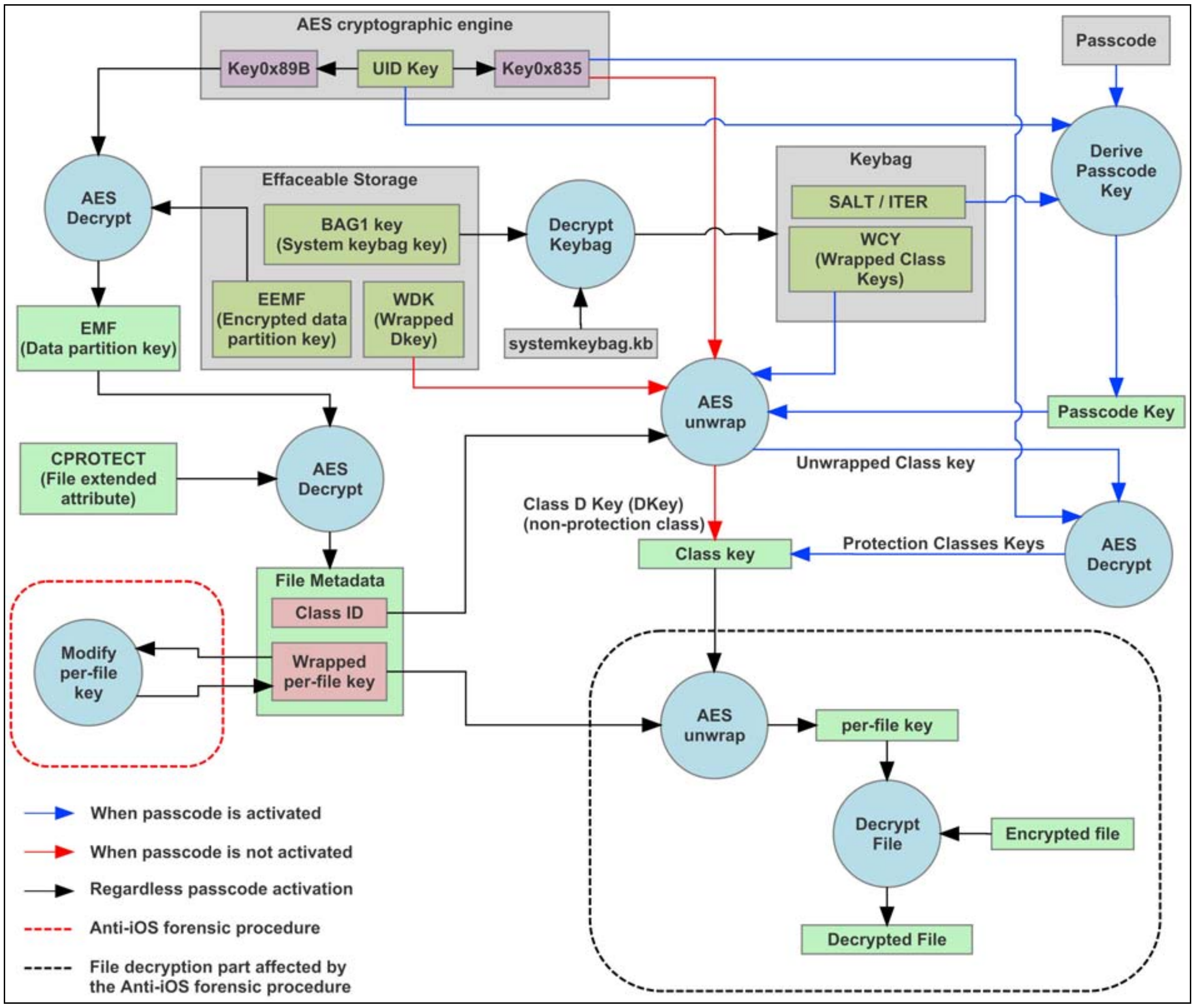


**Figure 1. An overview of iOS file decryption process**
Source: The first author's compilation

**Table 2. Anti iOS forensic toolkit**

| No. | Item |
|---|---|
| 1. | redsn0w 0.9.9b5. |
| 2. | Our command-line applications installed in the custom RAM disk – a block of RAM (primary storage or volatile memory):<br>• setkey (our application for data and per-file-key modification)<br>  ○ '-h' is the input parameter to render file contents unreadable (see subsection 2.4.1), and<br>  ○ '-d' is the input parameter to render the file contents unrecoverable (see subsection 2.4.2).<br>• setdate (our application for file-date modification)<br>• dumpjournal (our application for extraction of the journal file) |
| 3 | Shell commands installed in the custom RAM disk: mount_hfs, chmod, chwon, ls, rm. |
| 4. | MacBook Pro laptop. |

## 2.1. Step 1: Preparation (Background knowledge)

This step plays an important role in determining the location and storage format of the cryptographic keys in order to conceal (render selected file contents unreadable), delete (render selected file contents unrecoverable) and insert data (e.g. to mislead any forensic investigations). To reduce the chances of being detected, it is crucial that the individual using our technique is familiar with the most recent developments in iOS devices (e.g. up-to-date and in-depth understanding of how file data is stored and secured on iOS devices). For instance in the situation where one would want to insert an image file or video clip taken with another device, it is important to note that images or video clips captured by an iOS device's camera are converted to the corresponding data format. The converted data are then formatted in a file container with a specific codec and encrypted with a unique 256-bit per-file key. As per RFC 3394 specification [15], the per-file key is then wrapped with a data protection class key and stored in the image/video file's metadata prior to being encrypted with the file system data partition key (*EMF key*) generated from the unique hardware UID [3]. The encrypted file is then stored in the device's flash memory.

It is generally acknowledged that iPhone is a secure electronic device with various security features that protect stored information. In addition, accessing any content can only be done through USB (Universal Serial Bus).

In this phase, it is essential to be aware of the following issues.

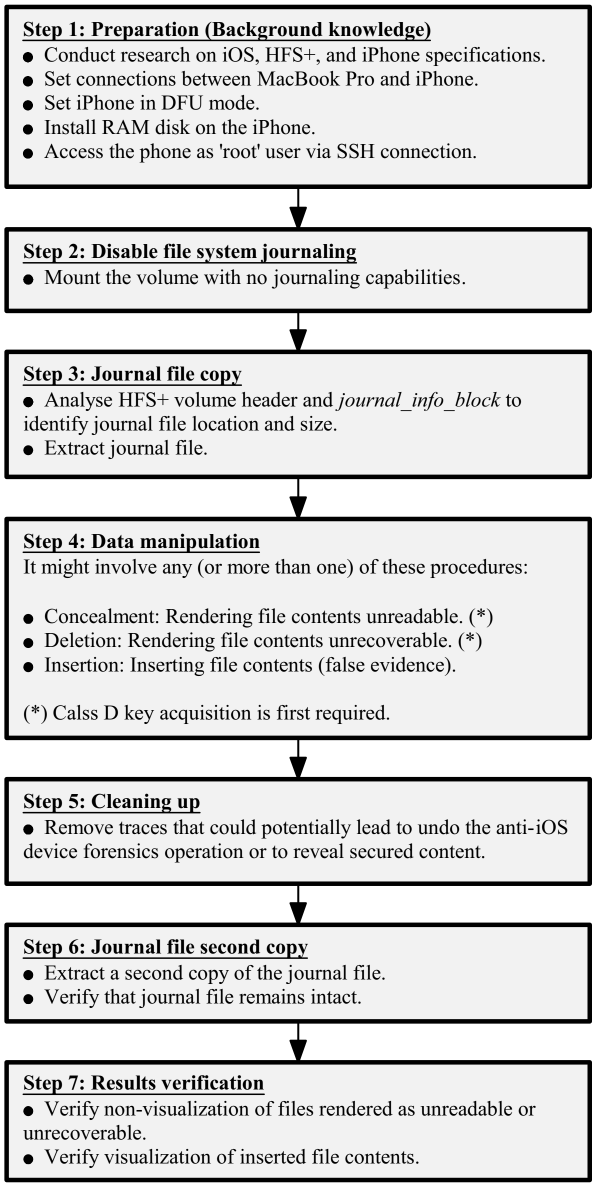
- Only iTunes application is natively allowed to operate via an USB connection on iPhones.
- Every file is encrypted in an iOS device (that supports encryption) and the encrypted file can only be decrypted using its per-file key, which is wrapped with the corresponding class key that the file is associated with.
- A per-file key is stored as an extended attribute of a file in its content protection (commonly cited as *cprotect*) structure allocated in the *Attributes* file of an HFS+ volume (see Figure 5).
- Passcode activation does not limit access to any types of files (except email messages) created by native applications. These files are secured with the non-protection class key, which wraps the per-file keys.
- No additional encryption derived from passcode protects either files generated by native applications or their per-file keys.
- With the exception of Class D data protection class key, protection class keys are then encrypted with a passcode-derived algorithm and stored in the *keybag*.
- Both the *keybag key* (BAG1) and the Class D key are stored in the Effaceable Storage (an area of the solid state NAND chip inside the device).
- UID (device's unique ID) is AES 256-bit key fused into the application processor during manufacturing.

As per the technique outlined in Figure 1, we inspected the iPhone and undertook the background research to have an in-depth understanding of its technical and complex encryption specifications. Based on the outcomes of that phase we were able to develop our own tools capable of running on a tether-jailbroken iPhone (once rebooted, the device does not maintain its jailbroken state and all applications no longer remain). The preliminary research also allowed us to determine what other necessary tools are to be included.

Adopting the method described by Zdziarski [9], a custom RAM disk was built to provide access functionalities, which combined with our iOS anti - forensic toolkit can then be used to perform the procedures described in this paper.

The device was then placed in DFU (Device Firmware Upgrade) mode by holding the Home and Power buttons. This was detected by redsn0w 0.9.9b5[2], which uses the DFU mode to exploit Boot ROM vulnerabilities, and injects its own code for kernel patching to bypass the iPhone signature verification. The patched kernel was not installed on the iPhone's flash memory and only operated in the device RAM, where redsn0w also installed and booted the custom RAM disk. This technique provides the capability to physically (block by block) or logically (mounting the device's file system) access the iPhone data area via the USB port and SSH (Secure Shell) connections. Once the connections are established as 'root' user and the custom RAM disk is booted, a command-line interface on the MacBook Pro is utilised for execution of programs on the iPhone.

[2] The jailbreaking tool is available for download on http://cydiahelp.com/redsn0w-0.9.9b5-download-available-brings-custom-ipsw-option/ [last accessed on 21 Aug 2013].



**Step 1: Preparation (Background knowledge)**
- Conduct research on iOS, HFS+, and iPhone specifications.
- Set connections between MacBook Pro and iPhone.
- Set iPhone in DFU mode.
- Install RAM disk on the iPhone.
- Access the phone as 'root' user via SSH connection.

**Step 2: Disable file system journaling**
- Mount the volume with no journaling capabilities.

**Step 3: Journal file copy**
- Analyse HFS+ volume header and *journal_info_block* to identify journal file location and size.
- Extract journal file.

**Step 4: Data manipulation**
It might involve any (or more than one) of these procedures:

- Concealment: Rendering file contents unreadable. (*)
- Deletion: Rendering file contents unrecoverable. (*)
- Insertion: Inserting file contents (false evidence).

(*) Calss D key acquisition is first required.

**Step 5: Cleaning up**
- Remove traces that could potentially lead to undo the anti-iOS device forensics operation or to reveal secured content.

**Step 6: Journal file second copy**
- Extract a second copy of the journal file.
- Verify that journal file remains intact.

**Step 7: Results verification**
- Verify non-visualization of files rendered as unreadable or unrecoverable.
- Verify visualization of inserted file contents.

**Figure 2. Our proposed iOS anti-forensic technique**

## 2.2. Step 2: Disable file system journaling

Journaling for the HFS+ volume is turned on by default in iOS devices, which logs all transactions (e.g. changes to the HFS+ volume). This feature helps to protect the file system against power outages or hardware component failures. Previous research has shown that transactions in the journal file could contain important metadata [4]. In the case of an iOS file system, metadata such as file encryption keys, and iOS anti-forensic activities such as ours would be logged by the journaling feature. Therefore, by disabling the journal, we believe that it will significantly reduce the chances of our technique being detected in a forensic investigation. For example, in our case study we need to mount the user data partition on the device's flash memory to logically access the extended attributes where per-file keys are stored. Therefore, to minimise leaving traces on the file system, we disable the file system journaling using the *mount_hfs* command:

```
mount_hfs -j /dev/disk0s1s2 /mnt2
```

The execution of the *mount_hfs* command ensures that none of our activities generates traceable transactions in the journal file. The user partition is represented by the parameter /dev/disk0s1s2, while /mnt2 indicates the RAM disk directory where the partition is mounted. Once the mount point provides access to the volume, *cprotect* attributes for all files are automatically decrypted by the OS if accessed – see Figure 3.

```
struct cp_xattr_v4 {
    u_int16_t xattr_major_version;
    u_int16_t xattr_minor_version;
    u_int32_t flags;
    u_int32_t persistent_class;
    u_int32_t key_size;
    u_int32_t reserved1;
    u_int32_t reserved2;
    u_int32_t reserved3;
    u_int32_t reserved4;
    u_int32_t reserved5;
    uint8_t   persistent_key[40];
};
```

**Figure 3. Cprotect extended attribute v4**

When the volume is mounted, files in the camera roll can be found in the '/mnt2/mobile/Media/DCIM/100APPLE/' directory.

## 2.3. Step 3: Journal file copy

Although the acquisition of the two copies of the journal file (before and after our activities) is not necessary, this allows us to verify whether the file system journaling had been disabled. For example, comparing the two different copies (or their hash values) allow us to determine whether any modifications have occurred. As the journal file has no entry in the file system (i.e. the journal file cannot be located using a file name search), we need to locate the file using the *journal_info_block* field of the volume header. The latter contains the number of the allocation block where the journal header is stored. Once the journal file location and size are found from the journal header, the volume has to be accessed physically and read block by block in order to copy the journal file from the iOS device to the MacBook Pro. As a consequence, Steps 2 and 3 of our proposed technique could be swapped since the volume does not necessarily have to be mounted in order to physically access blocks.

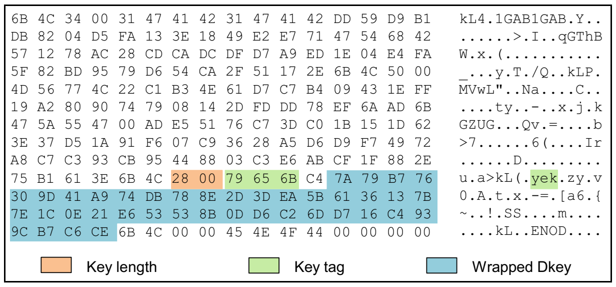
To ensure that the journal file is truly disabled while undertaking any iOS anti-forensic activities, we compared the “before” and “later” states that were copied using our *dumpjournal* application. The application allows users to extract the journal file directly from a volume (instead of extracting from a physical copy of the partition) and a copy is saved on the MacBook Pro.

## 2.4. Step 4: Data manipulation

The procedures presented in subsections 2.4.1 and 2.4.2 enhance the protection of sensitive data that is currently not protected by the 4-digit passcode or alphanumeric password in iOS devices, using our *setkey* application (see Table 2). The latter internally calculates the *Dkey* as described below. Subsection 2.4.3 describes our data insertion procedure.

### ➢ Class D key acquisition

The *Dkey* is stored in the Effaceable Storage but prior to use, it has to be unwrapped with the *Key0x835* (the device key derived from the UID at boot time) as described in Figure 1. By communicating with the kernel via specific connection methods provided with the Apple’s IOKit (*IOConnectCallMethod and IOConnectCallStructMethod*), *setkey* is capable of reading the Effaceable Storage content (see Figure 4) and the *Key0x835*.



```
6B 4C 34 00 31 47 41 42 31 47 41 42 DD 59 D9 B1   kL4.1GAB1GAB.Y..
DB 82 04 D5 FA 13 3E 18 49 E2 E7 71 47 54 68 42   ......>.I..qGThB
57 12 78 AC 28 CD CA DC DF D7 A9 ED 1E 04 E4 FA   W.x.(...........
5F 82 BD 95 79 D6 54 CA 2F 51 17 2E 6B 4C 50 00   _...y.T./Q..kLP.
4D 56 77 4C 22 C1 B3 4E 61 D7 C7 B4 09 43 1E FF   MVwL"..Na....C..
19 A2 80 90 74 79 08 14 2D FD DD 78 EF 6A AD 6B   ....ty..-..x.j.k
47 5A 55 47 00 AD E5 51 76 C7 3D C0 1B 15 1D 62   GZUG...Qv.=....b
3E 37 D5 1A 91 F6 07 C9 36 28 A5 D6 D9 F7 49 72   >7......6(....Ir
A8 C7 C3 93 CB 95 44 88 03 C3 E6 AB CF 1F 88 2E   ......D.........
75 B1 61 3E 6B 4C 28 00 79 65 6B C4 7A 79 B7 76   u.a>kL(.yek.zy.v
30 9D 41 A9 74 DB 78 8E 2D 3D EA 5B 61 36 13 7B   0.A.t.x.-=.[a6.{
7E 1C 0E 21 E6 53 53 8B 0D D6 C2 6D D7 16 C4 93   ~..!.SS....m....
9C B7 C6 CE 6B 4C 00 00 45 4E 4F 44 00 00 00 00   ....kL..ENOD....
```

**Figure 4. Wrapped *Dkey* in effaceable storage**

Both *Key0x835* and Class D key (DKey) are calculated in accordance to the functions presented in Table 3 and the AES key unwrapping specifications (RFC 3394). However, the *Key0x835* is computed and returned by the kernel as the UID is a hardware key embedded in the AES engine.

**Table 3. Notations and functions**

| | |
|---|---|
| **AES** $_{K}(D)$ | Encrypt *D* using the AES codebook with key *K* |
| **AES** $_{K}^{-1}(W)$ | Decrypt *W* using the AES codebook with key *K* |
| *K* | The encryption / decryption key |
| *W* | Data to be unwrapped (a wrapped key for this case study) |
| *D* | Data to be wrapped (a per-file key for this case study) |
| *B1* $\oplus$ *B2* | The bitwise exclusive or (XOR) of B1 and B2 |

- *Key0x835* = **AES** $_{\mathbf{UID}}$ (‘01010101010101010101010101010101’)

```
Key0x835 = [56 E5 D9 57 39 A1 5C 95 65 A7 9F 00 4B AC CF 6A]
```

- *Dkey* = **AES** $_{\mathbf{Key0x835}}^{-1}$ (*Wrapped Dkey*)

```
Dkey = [6B 17 C3 53 59 15 84 7E 6A 13 8D 6A 1C 9F 65 61
        B4 DA 9D 35 EA 5A E1 4F E5 A7 35 55 14 F3 A2 EE]
```

The Class D key is then used to unwrap the per-file key associated with the targeted file.

In our concealment and deletion procedures, we will respectively modify the per-file key either using a reversible algorithm or wiping with zeros prior to wrapping the modified per-file key with the Class D key. We will then store the wrapped modified per-file key in the *cprotect* attribute, which will replace the original wrapped per-file key – see subsections 2.4.1 and 2.4.2.

**2.4.1. Concealment: Rendering file contents unreadable**. In this section, we demonstrate how we can conceal the image file IMG_0001 using our *setkey* application:

```
./setkey -h /mnt2/mobile/Media/DCIM/100APPLE/IMG_0001.JPG
```

This indicates that the per-file key stored in the IMG_0001.JPG’s *cprotect* extended attribute will be modified by applying a reversible mathematical calculation as described in this subsection. This procedure can be used, for example, to prevent non-authorised access to a file or to circumvent digital forensic investigations. The technique is carried out through system calls that Apple provides in order to get and set extended attribute values such as the *cprotect* attribute. These system calls are *getxattr* and *setxattr* (for more information, refer to [5] and [7]).

When processing IMG_0001.JPG, *setkey* reads its *'com.apple.system.cprotect'* attribute data into a buffer containing the *cprotect* structure (see Figure 3) described in section 2.2. For our case study, the values obtained are outlined in Figure 5.

```
04 00 00 00 08 00 00 00 04 00 00 00 28 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 90 6B 77 EB 0B A4 D6 CC 63 2A 96 CB
92 76 39 53 55 17 7A 7B 0B AB 30 86 E0 CD 8E 86
A0 2A D9 AD 49 53 98 E6 BE 06 AE 73
```

⬜ Major version 🟩 Class ID 🟧 Key length

🟨 Minor version 🟦 Wrapped per-file key

**Figure 5. IMG_0001's cprotect attribute data**

To reduce the risk of the device going in recovery mode, we do not directly modify the wrapped per-file key. Instead, the wrapped per-file key that is contained in the *cprotect* attribute (see Figure 5) is unwrapped, and the resulting per-file key is then modified. As IMG_0001.JPG belongs to Class ID #4 (non-protection Class D), the per-file key is unwrapped with the *Dkey*, as we have described in section 2.4, and we obtain the per-file key, $K_1$:

$$K_1 = \mathbf{AES}_{\mathbf{Dkey}}^{-1}\ (\textit{Wrapped per-file key})$$

```
K1 = [53 42 C2 C9 5F 73 3A C5 58 B4 AC 21 B2 CE 81 A2
      53 A9 29 09 AC 2B 97 B2 48 B3 E2 55 6D 7B D7 86]
```

The modification of $K_1$ will prevent IMG_0001.JPG from being read. Since this operation is intended to be reversible, the original key has to be recoverable. Therefore we use bitwise exclusive OR (XOR) with a 256-bit secret constant C[3] to modify $K_1$.

```
C = [89 F7 AB 7A BF 4B 77 DD 7B AB 7F CA C8 FD 57 E8
     8A 04 09 7B B8 F8 15 22 EA 6E B2 85 5C 1B 39 06]
```

The resulting per-file key is named $K_2$.

$$K_2 = K_1 \oplus C$$

```
K2 = [DA B5 69 B3 E0 38 4D 18 23 1F D3 EB 7A 33 D6 4A
      D9 AD 20 72 14 D3 82 90 A2 DD 50 D0 31 60 EE 80]
```

Since the constant *C* is only known to us (i.e. stored in our tool on the RAM disk), reversing this computation would not be possible by a third party (e.g. forensic practitioners working on the case) unless they are able to break the symmetric 256-bit key (e.g. a brute force attack is currently computationally hard).

In order to adhere to Apple specifications, the modified per-file key, $K_2$, is wrapped with the *Dkey* and the result ($WK_2$) written in the IMG_0001.JPG's *cprotect* attribute (see Figure 6).

$$WK_2 = \mathbf{AES}_{\mathbf{Dkey}}\ (K_2)$$

Once the phone is rebooted after this procedure, the image file (IMG_0001.JPG) will no longer be displayed. To recover IMG_0001.JPG, we would need to run our *setkey* application that would restore the original per-file key, and hence rendering the file content readable again as the XOR operation is self-inverse.

```
04 00 00 00 08 00 00 00 04 00 00 00 28 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 3D 5E A3 27 18 BE A5 78 51 FB 9E 6B
C3 3B 4D 6F C1 12 B4 81 AA 68 EE 0A 2A 58 89 F1
37 20 A7 72 DA D7 57 E3 97 2A 49 91
```

⬜ Major version 🟩 Class ID 🟧 Key length

🟨 Minor version 🟪 Wrapped concealed-per-file key (*WK2*)

**Figure 6. IMG_0001's cprotect attribute data containing $WK_2$**

**2.4.2. Deletion: Rendering file contents unrecoverable**. Similar to the concealment procedure in the preceding subsection, we use the *setkey* application to delete per-file keys associated with the selected file contents (instead of modifying the per-file keys). The only difference is that IMG_0001.JPG's per-file key ($K_1$) is being wiped with zeros rather than being concealed with a constant *C*.

For key deletion, we input the '-d' parameter in the command line when running the *setkey* application:

```
./setkey -d /mnt2/mobile/Media/DCIM/100APPLE/IMG_0001.JPG
```

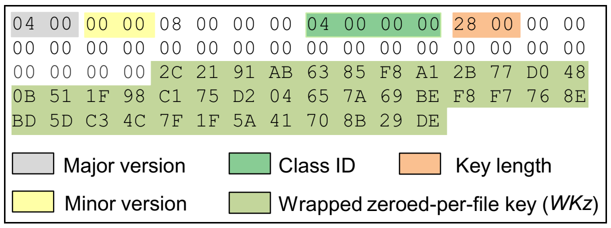
After reading IMG_0001.JPG's *cprotect* attribute data, the wrapped per-file key is replaced by *WKz* – the result of wrapping a zeroed 256-bit key with the *Dkey*.

$$WKz = \mathbf{AES}_{\mathbf{Dkey}}\ (Kz)$$

```
WKz = [2C 21 91 AB 63 85 F8 A1 2B 77 D0 48 0B 51 1F 98
       C1 75 D2 04 65 7A 69 BE F8 F7 76 8E BD 5D C3 4C
       7F 1F 5A 41 70 8B 29 DE]
```

[3] For simplicity, we use a secret constant, C, in our case study. In real life deployment, this value should be a random 256-bit string. For our "Concealment" procedure, the random 256-bit string would need to be stored in some databases to facilitate recovery at a later stage.

The wrapped zeroed-per-file key (*WKz*) is then copied to IMG_0001.JPG's *cprotect* attribute to ensure that the modification is irreversible (see Figure 7). Even though the particular file still exists in the data partition, it can no longer be decrypted as its original per-file key was zeroed.



```
04 00 00 00 08 00 00 00 04 00 00 00 28 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 2C 21 91 AB 63 85 F8 A1 2B 77 D0 48
0B 51 1F 98 C1 75 D2 04 65 7A 69 BE F8 F7 76 8E
BD 5D C3 4C 7F 1F 5A 41 70 8B 29 DE
```

⬜ Major version 🟩 Class ID 🟧 Key length
🟨 Minor version ▨ Wrapped zeroed-per-file key (*WKz*)

**Figure 7. IMG_0001's cprotect attribute data containing *WKz***

This procedure will prevent forensic practitioners from recovering the selected content or the relevant per-file key.

**2.4.3. Insertion: Inserting file contents (false evidence).** This procedure is independent from the procedures described in subsections 2.4.1 and 2.4.2. The aim is to insert false evidence that could potentially mislead a forensic investigation.

Inserting file created with another iOS device into the target iOS device without any tell-tale sign is a delicate procedure. For example, to insert a photo in the camera roll of our case study iPhone and to minimise the risk of this procedure being detected during forensic examination, we have to consider the following:

- The file naming convention and location where such files are saved;
- The file encoding;
- Modifying the file system metadata once the photo is inserted into the HFS+ volume.

➢ **File name and location**

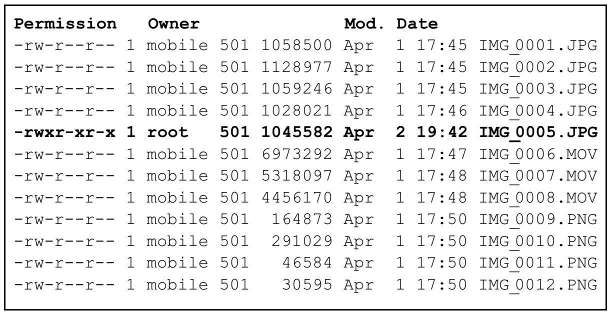
As explained in subsection 2.2, photos and videos captured with the iPhone's camera as well as screenshots are saved in the '/mobile/Media/DCIM/100APPLE/' directory where filenames are in the 'IMG_XXXX.EXT' nomenclature. 'XXXX' is a sequential number and 'EXT' is the file extension (JGP, MOV, or PNG). Understanding the file naming convention is essential when inserting file contents into the selected iOS device. For example, inserting a file '0013.JPG' (instead of IMG_0013.JPG) would be a tell-tale sign that the device has been tampered with. In our case study, we named the inserted file IMG_0005.JPG and replaced the existing file in order to show the importance of modifying dates (see next subsection).

➢ **Internal encoding of the inserted file**

Metadata of a JPEG file includes information about the camera model and brand, timestamp, Exif (Exchangeable image file format) and software version, Exif byte order, image and thumbnail size, resolution, etc. Therefore, to insert a file into another device requires an in-depth understanding of the file type and its metadata, as modifying certain values might result in the file not been able to be read by the device. Therefore, to minimise the risks of the inserted file being detected by a forensic practitioner, we recommend that the file to be inserted is created using hardware with a similar specification. In our case study, we inserted a photo taken with another iPhone 4 with a similar specification.

➢ **False evidence implantation**

Using an SSH connection via the USB port, the external IMG_0005.JPG was copied from the MacBook Pro to the camera roll directory in the mounted HFS+ volume of the iPhone's user data partition (/mnt2/mobile/Media/DCIM/100APPLE/). Although doing so replaces the previous file, this activity was picked up in the console terminal – see Figure 8:



```
Permission   Owner              Mod. Date
-rw-r--r-- 1 mobile 501 1058500 Apr  1 17:45 IMG_0001.JPG
-rw-r--r-- 1 mobile 501 1128977 Apr  1 17:45 IMG_0002.JPG
-rw-r--r-- 1 mobile 501 1059246 Apr  1 17:45 IMG_0003.JPG
-rw-r--r-- 1 mobile 501 1028021 Apr  1 17:46 IMG_0004.JPG
-rwxr-xr-x 1 root   501 1045582 Apr  2 19:42 IMG_0005.JPG
-rw-r--r-- 1 mobile 501 6973292 Apr  1 17:47 IMG_0006.MOV
-rw-r--r-- 1 mobile 501 5318097 Apr  1 17:48 IMG_0007.MOV
-rw-r--r-- 1 mobile 501 4456170 Apr  1 17:48 IMG_0008.MOV
-rw-r--r-- 1 mobile 501  164873 Apr  1 17:50 IMG_0009.PNG
-rw-r--r-- 1 mobile 501  291029 Apr  1 17:50 IMG_0010.PNG
-rw-r--r-- 1 mobile 501   46584 Apr  1 17:50 IMG_0011.PNG
-rw-r--r-- 1 mobile 501   30595 Apr  1 17:50 IMG_0012.PNG
```

**Figure 8. Detailed view of files in the camera roll**

We would now need to modify IMG_0005.JPG's file system metadata – access permissions, owner name, and creation, modification, and file access dates/times.

*Access permissions*

File access permissions for the inserted IMG_0005.JPG (-rwxr-xr-x) were changed to -rw-r--r—(the same as they were set for other files) using the Unix-like *chmod* command (on our RAM disk):

```
chmod 644 /mnt2/mobile/Media/DCIM/100APPLE/IMG_0005.JPG
```

*Owner name*

Since the iPhone was booted as 'root' user, the owner name for the inserted file needs to be modified to 'mobile' (default name used by iPhone), which was achieved by using the *chown* command:

```
chown mobile /mnt2/mobile/Media/DCIM/100APPLE/IMG_0005.JPG
```

*File date/time*

We used our *setdate* application (that executes Apple's system calls – *getattrlist* [6] and *setattrlist* [8]) to modify the creation, insertion and modification dates and times of the inserted IMG_0005.JPG. The timestamp of the inserted IMG_0005.JPG is now consistent with the other image files on the iPhone:

```
-rw-r--r-- 1 mobile 501 1028021 Apr  1 17:46 IMG_0004.JPG
-rw-r--r-- 1 mobile 501 1045582 Apr  1 17:47 IMG_0005.JPG
-rw-r--r-- 1 mobile 501 6973292 Apr  1 17:47 IMG_0006.MOV
```

## 2.5. Step 5: Cleaning up

Once any of the procedures in subsections 2.4.1 to 2.4.3 have been completed, we would need to remove any remnants that could reveal traces of our technique.

In our case study involving image files, it is important to note that thumbnails (reduced-size versions) of images are separately stored on the device. Therefore modifying a per-file key would only render the corresponding file content unreadable or unrecoverable. Since the image file still exists in the file system, the thumbnail will remain in the camera roll (120x120 and 158x158 pixels for iPhone 4).

The thumbnail located in the '/mnt2/ /mobile/Media/PhotoData/Thumbnails/' directory is then deleted using the *rm* command.

```
rm /mnt2/mobile/Media/PhotoData/Thumbnails/*
```

## 2.6. Step 6: Journal file second copy

We now execute the *dumpjournal* application again to obtain a second copy of the journal file. Both copies of the journal file (or their hash values) obtained from Step 3 and this step corroborate that all three of our procedures were not captured by the journaling feature (i.e. both journal files have the same hash values, as no transactions were recorded since the journaling feature was disabled in Step 2 – see Section 2.2).

## 2.7. Step 7: Results verification

In this step, we disconnect, turn off and reboot the iPhone. The file system journaling will automatically be enabled when the iPhone is rebooted.

## 3. Findings

As shown in Figure 9, after applying the concealment or the deletion technique to all files in the camera roll, the rebooted iPhone did not display the images and video files whose per-file keys had been modified. Only the default image indicating the file type was displayed (and the thumbnails for the 12 images and video files were not available).

As shown in Figure 10, we have successfully replaced the original IMG_0005.JPG with another image using our file insertion procedure.

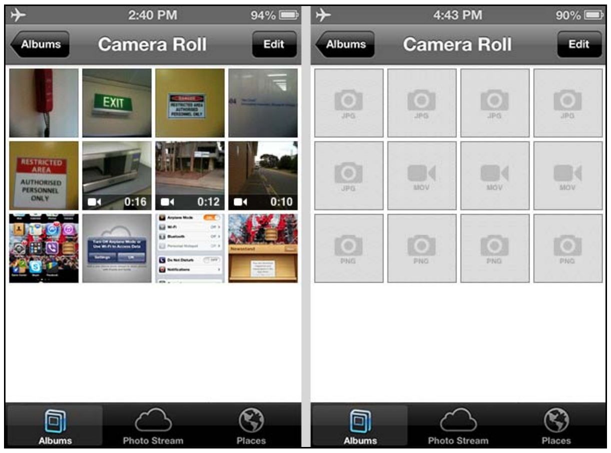


**Figure 9. Camera roll before (left) and after (right) the modification of per-file keys**

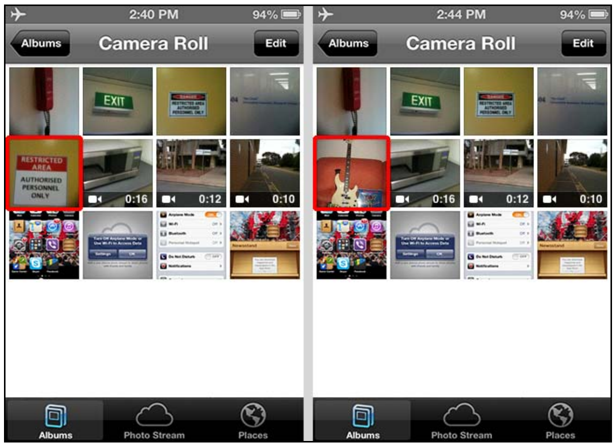


**Figure 10. Camera roll before (left) and after (right) the file insertion procedure**

Our procedures were not captured in the journal file and no information was recoverable from the iPhone that could be used to recover the respective per-file keys to reverse our procedures. This supports our observation that disabling file system journaling plays a vital role in reducing traceable trails.

## 4. Conclusions and future work

In this paper, we demonstrated that our concealment technique can be used to prevent the disclosure of contents currently not protected by iOS passcode/password activation. In addition to non-protected (Class D) data, our concealment and deletion techniques[4] can be used to secure data associated with protection classes A, B, and C (including data on iOS devices vulnerable to the lock screen bypass flaw [16]). Our techniques can be used to complement Apple's existing data protection mechanisms to protect data stored on iOS devices.

We also demonstrated that the insertion of an external file in the file system structure is feasible, and disabling file system journaling is important in anti-forensic procedures. It is also important to note that when inserting file contents in an iOS device, one would have to pay attention to the file metadata and make the necessary modifications to minimise the risks of such activities being detected during forensic examinations.

Future research includes improving the "Concealment" and "Deletion" techniques to better obfuscate activities such as per-file key modification and deletion, and examining how we can manipulate cp_xattr_v4 structure without setting the iOS device into recovery mode.

[4] Both procedures would only need to be slightly modified to wrap a concealed (or wiped) per-file key with the corresponding protection class key (instead of the Class D key).

## 5. References

[1] Azadegan, S., Yu, W., Liu, H., Sistani, M. & Acharya, S. 2012. Novel anti-forensics approaches for smart phones. In Proceedings of HICSS 2012, pp. 5424–5431.

[2] Apple. 2004. Hfs plus volume format. Technical Note TN1150. DOI=http://developer.apple.com/legacy/mac/library/#technotes/tn/tn1150.html

[3] Apple.2012. iOS security (Oct. 2012). DOI=http://images.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf

[4] Burghardt, A. & Feldman, A. 2008. Using the HFS+ journal for deleted file recovery. Digital Investigations, 5 (Supplement), pp. S76–S82.

[5] MAC Developer Library. getxattr(2) OS developer tools manual page. DOI= https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/getxattr.2.html

[6] MAC Developer Library. getattrlist(2) OS developer tools manual page. DOI= https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/getattrlist.2.html

[7] MAC Developer Library. setxattr(2) OS developer tools manual page. DOI= https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/setxattr.2.html

[8] MAC Developer Library. setattrlist(2) OS developer tools manual page. DOI= https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/setattrlist.2.html

[9] Zdziarski, J. 2012. Hacking and securing iOS applications. Sebastopol: O'Reilly Media, Inc.

[10] Distefano, A., Mea, G. & Pace, F. 2010. Android anti-forensics through a local paradigm. Digital Investigation, 7(Supplement), pp. S83–S94.

[11] Hilley, S. 2007. Anti-forensics with a small army of exploits. Digital Investigation, 4(1), pp. 13–15.

[12] Kianas, A., Restivo, K., & Shirer, M. 2012. Android and iOS surge to new smartphone OS record in second quarter, according to IDC. Press release 8 August. http://www.idc.com/getdoc.jsp?containerId=prUS23638712

[13] Quick, D. & Choo, KKR. 2013a. Dropbox analysis: Data remnants on user machines. Digital Investigation, 10(1), pp. 3–18.

[14] Quick, D. & Choo, KKR. 2013b. Digital droplets: Microsoft SkyDrive forensic data remnants. Future Generation Computer Systems, 29(6), pp. 1378–1394.

[15] Schaad, J. & Housley, R. 2002. RFC 3394: Advanced encryption standard (AES) key wrap algorithm, September. http://www.ietf.org/rfc/rfc3394.txt

[16] Whittaker, Z. 2013. Apple iOS 6.1.3 fix contains another lock screen bypass flaw. ZDNet, 20 March. http://www.zdnet.com/apple-ios-6-1-3-fix-contains-another-lock-screen-bypass-flaw-7000012912/

[17] Tassone, C., Martini, B., Choo, KKR., Slay, J. 2013. Mobile device forensics: A snapshot. Trends & Issues in Crime and Criminal Justice, 460, pp. 1–7.