Exploring the iPhone Backup made by iTunes

[Authors' names]

[institution]
[mail address]

Abstract

The iPhone mobile from Apple Inc. is one of the most notable phones on the market thanks to its simple and user-friendly interface and ever growing pool of available high quality applications for both personal and business use. The increasing use of iPhone mobiles leads forensics practitioners towards the need for tools to access and analyze the information stored in the device. This research aims at describing how to forensically analyze a logical backup of an iPhone made by the Apple iTunes utility, understanding its structure and creating a simple tool to automate the process of decoding and analyzing the data. It was found that significant data of forensic value such as e-mail messages, text and multimedia messages, calendar events, browsing history, GPRS locations, contacts, call history and voicemail recordings can be retrieved using this method of iPhone acquisition.

Keywords: iPhone, iTunes, iOS, Logical Backup, Mobile Phone Forensics, iPBA, iPhone Backup Analyzer.

1 Introduction

Modern mobile phones store vast amounts of data and have become an integral part of peoples' daily lives. The ever evolving technologies in the field of mobile communications introduced a whole new experience of using mobile devices, either for personal use or for work. People relies on smartphones for an infinite number of tasks, from planning their day to browsing the Internet. But the increasing usage of smartphones in day-to-day activities led these devices to store more and more information about their owners' life: where they've been, who did they call, who did they text, what did they do and so on. These pieces of data could be a rich source of evidence when the device itself is involved in a criminal activity and is seized as part of an investigation process, whether it is a target, a mean or just a silent "witness". This brings the need to study forensically sound methods to handle the examination and analysis of these devices and of the data they contain.

Among the mobile phones available on the market we chose to work on the iPhone device from Apple inc. The iPhone is a new generation smartphone which is seeing an incredible diffusion throughout the world, with a wide range of functions carrying a wide range of elements which can be extracted during an analysis. We decided to analyze the iPhone data via the backup feature, because that is the methodology which appears more forensically sound by producing the slightest amount of modifications in the device under test. While the acquisition of the content of an iPhone using its standard backup feature is widely known and reported, we found a lack of literature explaining in detail how the elements acquired this way are organized and which forensically interesting data they contain. This research will explore the logical backup of an iPhone made using the standard backup features and try to describe all its contents we've been able to identify and which could be useful during an investigation. We'll present also a tool we developed to browse through the elements of the backup.

This document is organized as follows:

- Section I: Introduction.
- Section II: Creating a logical backup. This section describes what is the logical backup of the iPhone, how can be obtained and why it can be useful for forensics purposes.

- Section III: Backup structure. This section describes the structure of the iPhone backup and the elements we found inside: the standard files and the possible types of the other ones.
- Section IV: Backup Content. This section gives a comprehensive list of all the data we found in the backup directory, with an explanation of how the single files and databases are structured and which useful information we could find in them.
- Section V: iPBA - iPhone Backup Analyzer. This section describes the software we built and used to analyze the backup data.
- Section VI: Conclusions.

2 Creating a logical backup

The backup data on which all of the further analysis is performed is obtained using the iTunes software. The iTunes utility by Apple inc. is available for Mac Os X and Windows platforms and is the default software (and the only officially supported one) to interact with the Apple iPhone. The software provides a backup feature which utilizes Apple's proprietary synchronization protocol to copy the iPhone data to a workstation. While the backup system can be manually disabled and provides an encryption function, the default behavior of iTunes is to make an unencrypted backup without asking, whenever the iPhone is connected. It can be noted that when the iPhone is synchronized with the computer it is paired with (via USB cable), data is copied from the phone to the computer and vice-versa, whether each element has a newer version on the phone or on the computer. Hence, in a forensics examination it is necessary to synch the device with a clean installation of iTunes which contains no data, to prevent newer elements to be copied from the forensics workstation to the device under analysis. Moreover it is known that the pairing can't be established via a USB write blocker, because the backup utility needs to mount the iPhone filesystem [Bader and Baggili, 2010]. While this methodology was initially developed to analyze seized mobile phones, it is noted that it can be used the same way to conduct forensics analysis on seized computers which have been paired with unknown iOS devices (being the backup feature enabled by default).

Our analysis was conducted on an iPhone 3GS with iOS version 4.2.1 and iTunes version 10.2, the latest versions available during the research work. While more extensive studies should be conducted to prove it, it is noted that all iPhone devices sharing the same version of the operating system should be equally interested by the results described in this document. Moreover, iPad devices, which share a slightly different version of iOS, should be analyzed with the same instruments.

The results we provide in this paper were achieved by studying the logical backup with iPBA *iPhone Backup Analyzer*, an utility we developed as a simple mean to browse through the backup data. The software is written in Python and provides a graphical interface which shows the tree of domains and files in the backup, and provides a simple analysis of each file based on the content of the file itself. For example, binary files of known types are automatically converted to a readable form. As another example, for each SQLite database file it shows a list of the tables it contains, and it provides the possibility to click on a table name to see its contents.

Deeper analysis of each file has been conducted with standard instruments which are described in the text. These instruments include some standard Mac Os X utilities (such as the *Plist editor* for analyzing binary and plain text plist files), third party utilities (such as the SQLite client to explore the content of SQLite database files) and standard UNIX command line utilities (such as *dd*, *strings* or *hexdump*).

3 Backup structure

The backup data is stored in a preconfigured folder for each of the operating systems iTunes is made available for.

Mac Os X: ~/Library/Application Support/MobileSync/Backup/

Windows XP: \Documents and Settings\(username)\Application Data\Apple Computer\MobileSync\Backup\

Windows Vista, Windows 7: \Users\(username)\AppData\Roaming\Apple Computer\MobileSync\Backup\

It is possible to have an arbitrary number of backups, as each backup is stored in a subdirectory of the previously described path. The name of the backed-up folder is a string of forty hexadecimal numbers and characters (0-9 and a-f), and represents a unique identifier for the device from where the backup was obtained. This unique identifier appears to be a hashed value since it was the same unique name given to the backed-up folder by iTunes on both Mac and Windows operating systems. Within this folder, there are hundreds of backup files with long hashed filenames consisting of forty numbers and characters. These filenames signify a unique identifier for each set of data copied from the iPhone memory [Bader and Baggili, 2010].

The files found in the backup directory can be classified into five categories:

- SQLite3 database files;
- Plain text plist files;
- Binary plist files;
- Multimedia and text files.
- Non-standard data files.

The SQLite3 database files store a single database each in SQLite3 format. Each database can contain an arbitrary number of tables. The plain text plist files are XML-like text files. Their binary counterparts are XML-like files stored in binary format which can be easily converted back to a plain text format with the Mac Os X *plutil* utility. A more in-depth description of those file types is provided in sections 3.2 and 3.3.

In addition to the files described before, the backup directory contains five more standard files with a fixed name, described in section 3.1.

3.1 Standard backup files

These files are created by the backup system and store data about the backup itself and the device it was made of. Their names are:

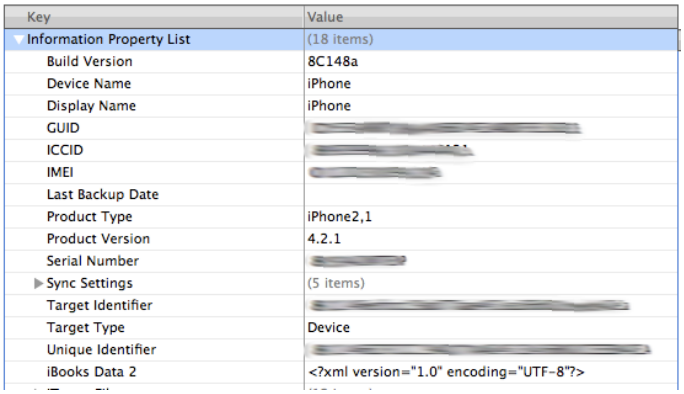
- *Info.plist*
- *Manifest.plist*
- *Status.plist*
- *Manifest.mbdb*
- *Manifest.mbdx*

The first three files are plist files which can be easily analyzed using the *Property List Editor* application on Mac Os X or by manual examination with a text editor (binary plist files must be converted to text format first by the MacOsX *plutil* utility). The file *Info.plist* (plain text plist) stores data about the backed up device (such as the device name, GUID[1], ICC-ID[2], IMEI[3], serial number) and the iTunes software used

[1] GUID (Global Unique IDentifier): a unique reference number used to identify a device.

[2] ICC-ID (Integrated Circuit Card IDentifier): an international identifier for the SIM card in the device.

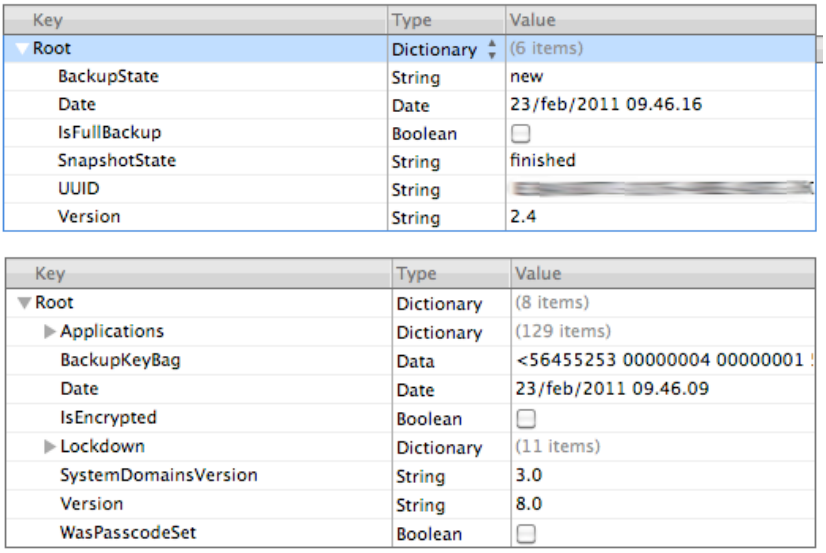
to build the backup (such as iTunes settings and the iTunes version number). A sample of the file as shown by the *Property List Editor* application is shown in figure 1.



Key	Value
Information Property List	(18 items)
Build Version	8C148a
Device Name	iPhone
Display Name	iPhone
GUID	[redacted]
ICCID	[redacted]
IMEI	[redacted]
Last Backup Date	
Product Type	iPhone2,1
Product Version	4.2.1
Serial Number	[redacted]
Sync Settings	(5 items)
Target Identifier	[redacted]
Target Type	Device
Unique Identifier	[redacted]
iBooks Data 2	<?xml version="1.0" encoding="UTF-8"?>

Figure 1: Info.plist shown by Property List Editor.

The file *Manifest.plist* (binary plist) describes the contents of the backup. In this file we find the applications installed on the backed up device (each with its version number), along with the date the backup was made, whether the backup is encrypted or not, and again data about the device and the iTunes software. A sample of the file as shown by the *Property List Editor* application is shown in figure 2.



Key	Type	Value
Root	Dictionary	(6 items)
BackupState	String	new
Date	Date	23/feb/2011 09.46.16
IsFullBackup	Boolean	☐
SnapshotState	String	finished
UUID	String	[redacted]
Version	String	2.4

Key	Type	Value
Root	Dictionary	(8 items)
Applications	Dictionary	(129 items)
BackupKeyBag	Data	<56455253 00000004 00000001 !
Date	Date	23/feb/2011 09.46.09
IsEncrypted	Boolean	☐
Lockdown	Dictionary	(11 items)
SystemDomainsVersion	String	3.0
Version	String	8.0
WasPasscodeSet	Boolean	☐

Figure 2: Manifest.plist and Status.plist shown by Property List Editor.

The file *Status.plist* (binary plist) seems to store information about the completion of the backup itself, such as whether the backup is complete or not. A sample of the file as shown by the *Property List Editor* application is shown in figure 2.

The last two files of the list, *Manifest.mdbd* and *Manifest.mbdx*, are binary files which store the descriptions of all the other files in the backup directory. It can be noted that in these files there are also, described as separate records, the directories and the symbolic links, which of course don't have a corresponding file in the backup directory.

---

[3] IMEI (International Mobile Equipment Identity): a number, usually unique, used to identify mobile phones.

The structure of the index file *Manifest.mbdx* is shown in figure 3.

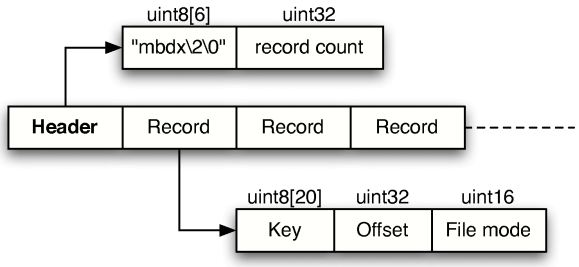


Figure 3: Structure of file Manifest.mbdx.

The file is made up by a header and a number of records, one for each element indexed (files, directories, symbolic links). The header contains two fields: a string identifying the file type and the number of records in file. Each record contains three fields. The first field is a 20 characters unique identifier of the element (if the element is a file, the key is also the name of the corresponding file stored in the backup directory). The second field is the offset (in bytes) of the corresponding element in the *Manifest.mbdb* file. It must be noted that offsets in file *Manifest.mbdb* don't count the file header (first 6 bytes), so to obtain the absolute offset we must add 6 to the value in the offset field. The last field is a 16 bit value describing the file permissions. The first 4 bits of this last field identify the file type:

- 0xAxxx: symbolic link
- 0x4xxx: directory
- 0x8xxx: regular file

The xxx part (three nibbles) in the file mode seems to carry information about the referenced element permissions in Unix style [2]. The three nibbles describe the permissions to read (R), modify (W) and execute (X) the file by the owner of the file, the users in the same group of the file and everyone else respectively, as shown in figure 4.

User				Group				Everyone			
	R	W	X		R	W	X		R	W	X

Figure 4: Structure of file permissions.

As an example, figure 5 shows the three nibbles for a file with a permissions field containing the hexadecimal value `0x8764`.

Example: a file with permissions 0x8764											
User				Group				Everyone			
	R	W	X		R	W	X		R	W	X
0	1	1	1	0	1	1	0	0	1	0	0
7				6				4			

Figure 5: Example of file permissions.

This file can be read, written and executed by the owner, read and written by the users in the same group of the file and only read by anyone else.

The second file, *Manifest.mbdb*, contains a record for each element indexed by the previous file. The structure of the file is shown in figure 6.

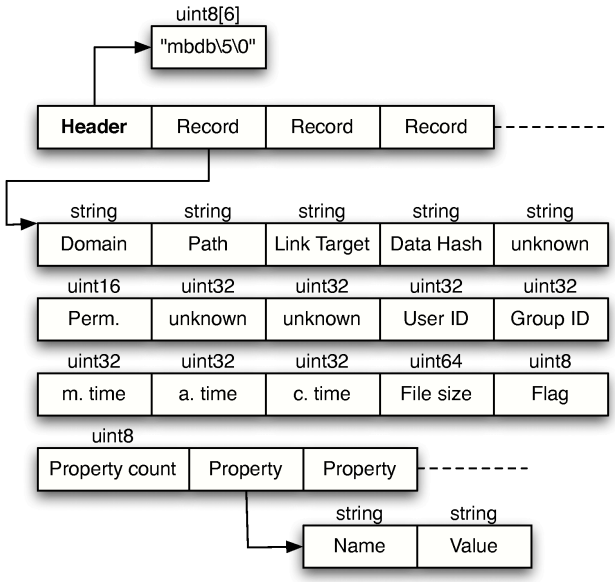


Figure 6: Structure of file Manifest.mbdb.

Each record can contain an integer, an array of integers or a string. The strings are composed of an uint16 that contains the length in bytes (or `0xFFFF` for empty strings) followed by the characters in UTF8 format (Unicode normalization form D). All the numbers are big endian.

As shown in figure, each record contains (among the other fields):

- *Domain*: the domain the element belongs to. Domains are a way to functionally categorize elements in the device backup and will be described later.
- *Path*: the full path of the element.
- *Link Target*: the target of the element, if the element itself is a symbolic link (otherwise the field contains value `0xFFFF`).
- *Mode*: the file permissions. This field holds the same value seen in file *Manifest.mbdx*.
- *User ID* and *Group ID*.
- *M. time*: the time (in Unix time format) when the actual content of the file was last modified.
- *A. time*: the time when the file was last accessed.
- *C. time*: the time when changes were last made to the file or directory's inode.
- *File size*: the size of the file in bytes.

Each record can contain a list of properties of arbitrary dimension [3].

3.2 SQLite files

SQLite is an ACID[4]-compliant embedded relational database management system contained in a relatively small (~ 275 kB) C programming library. The source code for SQLite is in the public domain and implements most of the SQL standard. It is arguably the most widely used database engine, as it is used today by several widespread browsers, operating systems and embedded systems among others. Due to its small size, SQLite is well suited to embedded systems, and is also included in Apple's iOS (where it represents one of the primary means of archiving data) [4].

For the purposes of this paper, the analysis of SQLite databases has been performed by a custom built software in Python. Nonetheless SQLite files can be studied by simply using the *sqlite* command line utility, which is available online[5] as open source software.

A database file can be opened by the command line:

```
$ sqlite3 filename
```

After the database has been opened, it is possibile to see the list of tables with the command:

```
$ .tables
```

The tables' contents can then be retrieved using standard SQL queries.

It is important to point out that the in some cases it is possible to retrieve deleted data from SQLite files. When a record is deleted, the area in the file where it was stored is marked as unused but is not immediately overwritten, so its content can already be found, for example by the Unix *strings* command. The *strings* command scans a file and outputs all the ASCII strings it found. It has been verified that among those strings it can be found the content of deleted records. By this mean it has been possible to recover, for example, deleted notes, messages and contact names from the device under test.

3.3 Plist files

Property Lists (often referred to as Plist files) are files that store serialized objects. In iOS (as well as all versions of Mac Os X from 10.0) the property lists are stored in XML format, with a public DTD[6] defined by Apple. The XML format supports non-ASCII characters and storing NSValue objects. The most used tags found on the device under test are:

- <*string*>: UTF-8 encoded string.
- <*real*>, <*integer*>: decimal string.
- <*true /*>, <*false /*>: boolean values (tag only, they don't contain other data).
- <*date*>: ISO 8601 formatted string representing a date.
- <*data*>: Base64 encoded data.
- <*array*>: a list that can contain an arbitrary number of elements, such as:

```
<array>
  <string> first </string>
  <string> second </string>
  <real> 23.44 </real>
</array>
```

- <*dict*>: a list containing an arbitrary number of pairs of <*key*> and plist element tags, such as:

```
<dict>
```

[4] ACID (atomicity, consistency, isolation, durability) is a set of properties that guarantee database transactions are processed reliably.

[5] http://www.sqlite.org/download.html

[6] DTD: Document Type Declaration

```
    <key> property1 </key>
    <string> value1 </string>
    <key> property2 </key>
    <real> 2.0 </real>
  </dict>
```

Since XML files are not the most space-efficient means of storage, Mac OS X 10.2 introduced a new format where property list files are stored as binary files. Starting with Mac OS X 10.4, this is the default format for preference files.

The *plutil* utility (introduced in Mac OS X 10.2) can be used to check the syntax of property lists, or convert a property list file from one format to another. XML property lists are hand-editable in any text editor, but Apple provides a "Property List Editor" application as part of their Developer Tools installation that provides a hierarchical viewer/editor which can also handle binary formatted plists. [5]

To decode a binary plist file using the *plutil* utility under Mac Os X we can use the command:

```
plutil -convert xml1 -o <outfile> <infile>
```

Base64 encoded data can be decoded with many utilities which can be freely found online. Mac Os X does not provide a default application to decode Base64, but the result can be achieved by saving the data block in a text file and then exploiting a function of the *openssl* utility (provided by default in the standard installation of Mac Os X):

```
openssl base64 -d -in <infile> -out <outfile>
```

4 Backup Content

The first categorization of backup files is described by their *domain*. The domain for each file is written in its corresponding record in the *Manifest.mbdb* file. Each file has a domain name chosen from the following list:

- Application domain.
- Home domain.
- Keychain domain.
- Managed Preferences domain.
- Media domain.
- Mobile Device domain.
- Root domain.
- System Preferences domain.
- Wireless domain.

The domains *Managed Preferences* and *Mobile Device* do not appear to contain useful informations (at least in the device under test), while the others contain useful data which is described in further sections. It is noted that elements in the *Application* domain are listed with a subdomain related to the name of the application they belong to, while elements in the other domains appear not to use this feature. When the subdomain is used, the domain string in *Manifest.mdbd* is written as `<domain>-<subdomain>`.

4.1 Application domain

The domain *AppDomain* contains a certain number of subdomains, one for each installed application. Each subdomain contains some files divided into a number of directories, most of them standard for all applications. A typical structure is shown in figure 4.1.

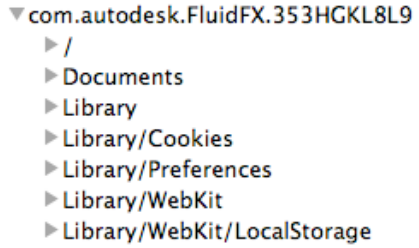


Figure 7: Typical structure of backup files of an application.

The directory *Documents* contains application-specific data, such as multimedia files for media players. Data in this directory is saved in application specific format.

The directory *Library* contains standard elements and has an almost identical structure for every application.

The subdirectory *Library/Cookies* contains a plist file named *Cookies.plist* containing the cookies for the application, i.e. simple data chunks used for temporary data storage.

The subdirectory *Library/Preferences* usually contains two files:

- *.GlobalPreferences.plist*: a symbolic link to a standard file containing settings common for all applications on the device.
- A file with the same name as the application with extension *plist* containing application specific preferences.

The subdirectory may also contain other settings files (usually symbolic links) related to standard elements of the operating system used in the application (such as *PeoplePicker.plist* for selecting names from the device Contacts or *ADlib.plist* for managing iAD banners).

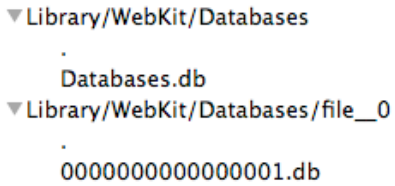


Figure 8: Webkit databases storage in backup data.

The subdirectory *Library/Webkit* contains elements related to Webkit, which is the standard engine used under iOS to render web pages. The directory contains a subdirectory *LocalStorage* into which temporary data is archived for offline use and could contain another directory named *Databases*. The ability of web applications to create databases on client machines is a new feature of HTML5 and the directory *Databases* is used to save such data. It has a structure similar to the one depicted in figure 8.

The file *Databases.db* is a SQlite database containing a table named *Databases* in which each record describes a Webkit database for offline data storage. For each record the table contains pieces of information such as the short name and the descriptive name.

`Fields of a sample table record:`

```
- guid : 1
- origin : file__0
- name : timelinedb
- displayName : Social Timeline Database
- estimatedSize : 5120
- path : 0000000000000001.db
```

For each Webkit database there is a subdirectory named as the *origin* field in the corresponding record of the table described before. In that directory we find a SQLite database file (named as the *path* field in the above mentioned record) which contains the data. The data structure in this database is application specific and can be easily analyzed to retrieve all the data stored in the Webkit offline storage by the application.

For example, Fring (a free voice-over-IP application) uses this methodology to store events (like call or chat logs or Twitter events) and contact photos. These pieces of information can be easily dumped from the database file and used as evidence. As an example we show a record from the *events* table describing a VOIP call.

```
- event_composite_id : CallEvent__0__0__2
- event_id : 2
- event_type : CallEvent
- addon_id : 0
- service_id : 3
- modified_timestamp : 2010-10-10T09:43:02Z
- text : Dialed fring Call.<br/> Call duration 00:08
- user_id : fring-test-call
- user_display_name : fring test call
- user_avatar : fring_testcall_avatar_small.png
- has_photos : false
- comments_count : 0
- event_service_icon : timeline_outgoing_call@2x.png
- likes_count : 0
- can_like : false
```

4.2 Home domain

The Home Domain contains a *Library* directory with a structure similar to what described for the single applications. This directory contains all data for the applications provided by default by iOS. In this section we describe the elements we have been able to identify, ordered by the application they belong to.

Address Book *Library/AddressBook* contains the address book data of the device. The directory contains two SQLite databases:

- *AddressBook.sqlitedb* contains contact data.
- *AddressBookImages.sqlitedb* contains contact images, both thumbnails and full size images.

The simplified structure of the address book database is shown in figure 9. The main table is the one named *ABPerson*, which stores a record for each element of the address book and the main values associated to it (first name, last name, organization, job title, creation date and so on). Each contact in the address book may have an unlimited number of contact data (phone numbers, emails, etc.), so these values are stored in a separate table *ABMultiValue*. Each record of this table has a label (chosen from table *ABMultiValueLabel*) and a value. Some kinds of entries can be made of more than one element (for example, the address field is made up of an address, a city name, a ZIP code and so on), so it's necessary

to link to another table *ABMultiValueEntry* where the single elements are stored, each one identified by a label from table *ABMultiValueEntryKey*.

Each entry in the address book can be linked to a group, defined in the table *ABGroup*. The connection between contacts and groups is made by the table *ABGroupMembers*.

At last, each contact may have an image. Images are stored in the database file *AddressBookImages.sqlitedb* as thumbnails (in table *ABThumbnaiImage*) and as full size images (table *ABFullSizeImage*) with crop size data.

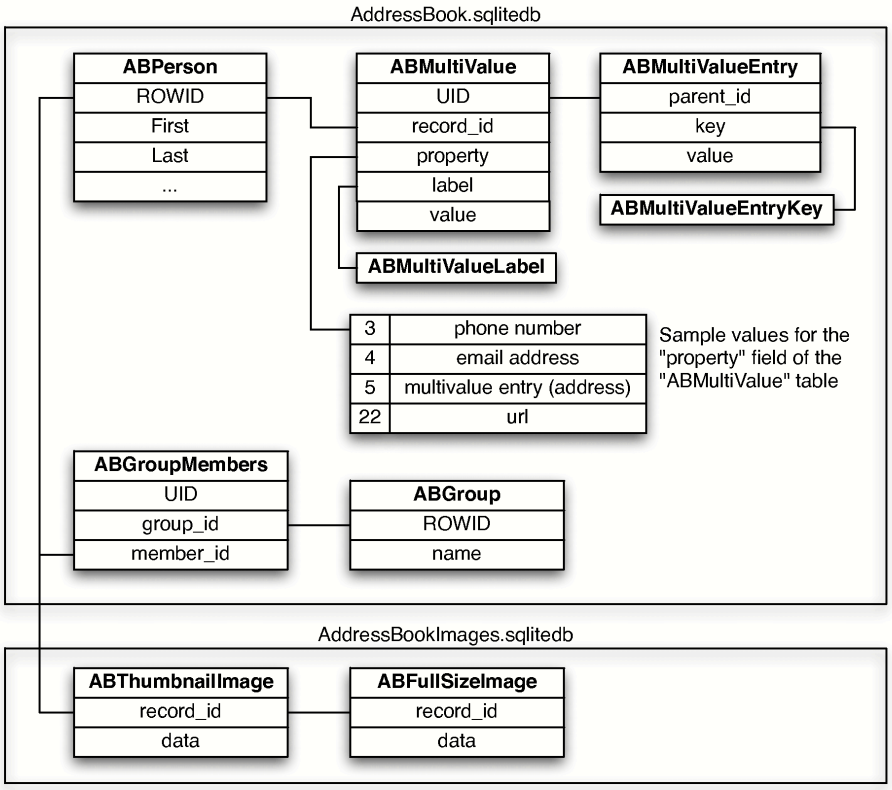


Figure 9: Structure of address book database files.

As an example we show the SQL queries which are used by the iPBA software to retrieve data from the address book database.

To retrieve groups:

```
SELECT ROWID, Name FROM ABGroup
```

To retrieve people for each group (identified by its index field ROWID):

```
[frame=single]
SELECT ABPerson.ROWID, First, Last, Organization FROM ABGroupMembers
INNER JOIN ABPerson ON ABGroupMembers.member_id = ABPerson.ROWID
WHERE ABGroupMembers.group_id = <ROWID> ORDER BY Last, First,
Organization
```

To retrieve the elements associated with a single person (identified by its index field ROWID):

```
[frame=single]
```

```
SELECT property, label, value, UID FROM ABMultiValue WHERE record_id =
<ROWID>
```

To retrieve rows for a multivalue element (ABMultiValue.property = 5) identified by its ABMultiValue.UID:

```
[frame=single]
SELECT KEY, value FROM ABMultiValueEntry WHERE parent_id = <UID> ORDER
BY key
```

Safari Mobile Browser *Library/Caches/Safari/Thumbnails* contains PNG images of the open tabs in the Safari browser. The files, along with their creation date and time, can provide useful information about the device and its user.

The directory *Library/Safari* also contains data related to the Safari web browser. Three files can be found in this directory:

- *Bookmarks.db* is a SQLite database file in which all the web bookmarks are stored. The database contains three tables, the most important of them being *bookmarks* which contains a record for each bookmark or collection of bookmarks (bookmark folder). An example of a bookmark record is shown below:

  ```
  - id : 4
  - special_id : 0
  - parent : 3
  - type : 0
  - title : Apple
  - url : http://www.apple.com/it/
  - num_children : 0
  - editable : 1
  - deletable : 1
  - order_index : 0
  - external_uuid : com.apple.sync services: 6B01EF49-4AEC-4B4B-
  BF84-6A7922499AAC
  ```

  Each record contains, among the other fields, a unique numeric id, a type identifier (0 for bookmarks and 1 for folders), a parent id (which links an element to the bookmark folder it is placed in), a descriptive title, the number of children (only for bookmark folders), the URL (only for bookmarks) and the attributes *deletable* and *editable* (1 for true and 0 for false).

- *History.plist* is a binary plist file which contains the recently visited web pages. It is structured as an array of dictionaries, each of one representing a single entry. Each element has, among the other attributes, a title (corresponding to the title of the linked page), an URL, the last visited date and a visits counter. An example is shown below:

  ```
  <dict>
  <key></key>
     <string>http://www.google.it</string>
  <key>D</key>
     <array>
        <integer>1</integer>
     </array>
  <key>lastVisitedDate</key>
     <string>315859729.5</string>
  <key>title</key>
     <string>Google</string>
  ```

```
<key>visitCount</key>
   <integer>1</integer>
</dict>
```

- *SuspendState.plist* is a binary plist file which contains information about the open tabs in the Safari browser. It is structured as an array of dictionaries (one for each tab), each one in turn containing a dictionary with information about the back function (i.e. the URLs visited before the current one in each tab). Every element of the outermost dictionary has a structure such as the one listed below.

```
<key>capacity</key>
   <integer>100</integer>
<key>current</key>
   <integer>4</integer>
<key>entries</key>
   <array>
      ... back function entries
   </array>
<key>SafariStateDocumentLastViewedTime</key>
   <real>319630534.65534902</real>
<key>SafariStateDocumentTitle</key>
   <string>Google</string>
<key>SafariStateDocumentURL</key>
   <string>http://www.google.it</string>
<key>SafariStateDocumentUUID</key>
   <string>985CDEAE-E8AB-4CEB-A1CC-6B78A8E8EECC</string>
<key>SafariStateDocumentWasOpenedFromLink</key>
   <false/>
```

  The above example shows the entry for the *Google* home page (url: *http://www.google.it*), which contains a dictionary (omitted key *entries*) of four entries (key *current*) representing the pages available for the back function. It also stores the last time the page was shown to the user (key *SafariStateDocumentLastViewedTime*).
  The omitted dictionary in the previous example contains, as stated before, four entries. Each entry has a structure similar to the one shown below:

```
<key></key>
   <string>http://www.google.it</string>
<key>WebViewportArguments</key>
   <dict>
      <key>height</key>
         <real>-1</real>
      <key>initial-scale</key>
         <real>-1</real>
      <key>maximum-scale</key>
         <real>-1</real>
      <key>minimum-scale</key>
         <real>-1</real>
      <key>user-scalable</key>
         <real>-1</real>
      <key>width</key>
         <real>-1</real>
   </dict>
<key>scale</key>
   <real>0.54387754201889038</real>
<key>scaleIsInitial</key>
```

```
   <false/>
<key>scrollPointX</key>
   <integer>97</integer>
<key>scrollPointY</key>
   <integer>7264</integer>
<key>title</key>
   <string>Google</string>
```

The above element is the page *Google* (url: *http://www.google.it*). The dictionary stores all the information needed to show the page as it was shown to the user last time (scale, vertical position, ...).

Calendar *Library/Calendar* contains the data for the Calendar application. The directory contains a single SQLite file named *Calendar.sqlitedb*. The simplified structure of the database is shown in figure 10. The main table in the database is the table *Event*, which stores a record for each calendar entry. This record is linked by the *calendar_id* field to the table *Calendar*, containing a list of the calendars available in the device, and by its primary key *ROWID* to the other tables. The most significant tables linked to a calendar entry are the one depicting the reminders (table *Alarm*), the recurrence of the event (daily, weekly, monthly, yearly, none) (table *Recurrence*) and the attendees to the event (listed in table *Participants* and linked to events via table *Attendees*).

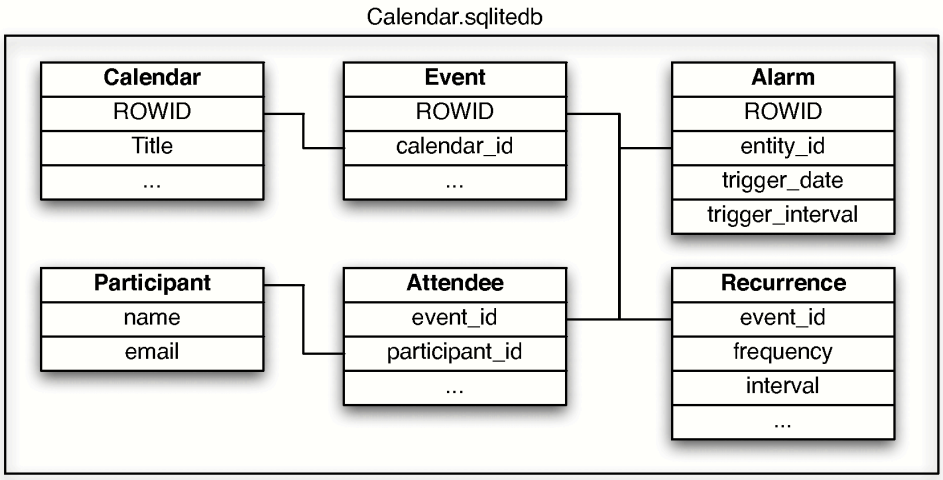


Figure 10: Structure of calendar database.

Configuration Profiles

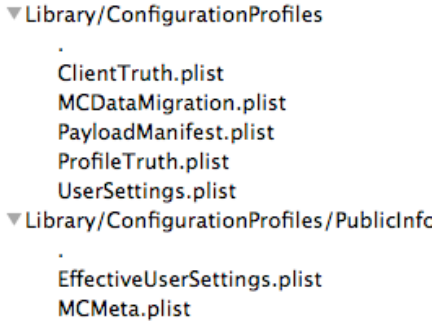


Figure : Contents of Configuration Profiles backup directory.

Library/ConfigurationProfiles and its subdirectory *PublicInfo* contain some plain text plist files related to the device configuration (the structure of this directory as found in the device under test is shown in figure 4.2). The most interesting files are *UserSettings.plist* and *EffectiveUserSettings.plist* which contain system preferences related to the capabilities of the device user (such as parental control). It is unknown what is the difference between the two aforementioned files, and they appear to contain the same values.

The two files show a structure in which the innermost element is a dictionary associated to the key *restrictedBool*, which in turn contains couples of keys and respective boolean values (contained in a single element dictionary), as shown below:

```
<plist version="1.0">
  <dict>
    <key>assignedObject</key>
    <dict/>
    <key>restrictedBool</key>
    <dict>
      <key>allowAccountModification</key>
      <dict>
        <key>value</key>
        <true/>
      </dict>
      <key>allowAddingGameCenterFriends</key>
      <dict>
        <key>value</key>
        <true/>
      </dict>
      <key>allowAppInstallation</key>
      <dict>
        <key>value</key>
        <true/>
      </dict>
```

Cookies

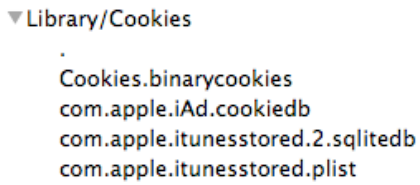


Figure 12: Contents of Cookies backup directory.

Directory *Library/Cookies* contains the device cookies, which are structures used to temporarily store small amounts of text. A cookie can be used for authentication, storing site preferences, shopping cart contents, the identifier for a server-based session, or anything else that can be accomplished through storing text data. A cookie consists of one or more name-value pairs containing pieces of information, which may be encrypted for information privacy and data security purposes.

In the analyzed device this directory contains four files:

- *Cookies.binarycookies*, a binary file which seems to contain cookies created by the Safari web browser. A simple string analysis using the *strings* Unix command revealed a list of visited sites

with their associated cookie values (which could include, for example, session keys that can be used for further investigation). As an example, the following listing contains part of the output of the Unix *strings* command performed on the previously described file.

```
.support.github.com
A__utmb
100728323.5.9.1292667876786
.support.github.com
A__utmz
100728323.1292667788.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(n
one)
.support.github.com
?__utma
73095439.847495668.1263931189.1263931189.1263931189.1
.manuali.net
?__utmv
73095439.user_level_anonymous
```

While it is not known what does the shown data mean, we can safely assume that the device browser accessed the listed urls: *support.github.com* and *manuali.net*. Further analysis could be performed on the single strings extracted to uncover their meaning and how they could be used during a forensic analysis (for example, a string bound to a url could be a session id for a logged in user; thanks to this information an investigator could be able to obtain more data about him from the provider of the web service).

- *com.apple.iAd.cookiedb*, a SQLite file which contains a single *cookies* table, presumably used to store cookies for the iAd system.
- *com.apple.itunesstored.plist*, a plain text plist file containing cookies stored by the iTunes Store as an array of dictionaries.

```
<dict>
  <key>Name</key>
     <string>Pod</string>
   <key>Value</key>
      <string>17</string>
   <key>Domain</key>
      <string>.apple.com</string>
   <key>Path</key>
      <string>/</string>
   <key>Expires</key>
      <date>2010-12-22T19:13:05Z</date>
   <key>Created</key>
      <real>312145992</real>
</dict>
```

- *com.apple.itunesstored.2.sqlitedb*, a SQLite file which also seems to contain cookies stored by iTunes Store. It contains a single *cookies* table, with records like the one below:

```
- discard : 1
- domain : .apple.com
- expire_time : 0.0
- name : mzf_in
```

```
- path : /WebObjects
- secure : 0
- user : 132550201
- value : 170303
- version : 0
```

Keyboard Directory *Library/Keyboard* contains one or more plain ASCII files used to store dynamic dictionary words, i.e. recently used words which are used by the auto-completion dictionary (as recently used words are supposed to be the most likely candidates for words auto completion). This list includes words not present in the default orthographic dictionaries which have been inserted by the user. In the analyzed device there are two files, one for each dictionary used (standard english dictionary and italian dictionary).

- *dynamic-text.dat*
- *it_IT-dynamic-text.dat*

These files can be examined by the Unix *strings* command to reveal all the words stored, and can be useful to retrieve non standard words used by the device owner, such as proper names.

```
$ strings 9117e9fe1450592488...

  DynamicDictionary-4
  locatelli
  Boario
  download
  pisogne
  packaging
  genealogy
  gettin
  geonav
  fringbox
```

Maps The Maps application provides an interface to the Google Maps service for searching and browsing locations. The directory *Library/Maps* contains data for the application in three binary plist files:

- *Bookmarks.plist* seems to contain the locations bookmarked in the application. After decoding the file with the *plist* utility, it appears to contain the bookmarks saved as *data* elements:

```
<plist version="1.0">
   <dict>
      <key>BookmarksData</key>
      <array>
      <data>
         CAAQABgDIAAqJ1ZpYSBWYWxsb
         NTA1MCBQYXNzaXJhbm8gQlMsI
         YWxpYTUBAAAAQglQYXNzaXJhb
         ...
      </data>
```

The bookmarks' data, stored under the *data* key, is encoded in Base64 format. After decoding we found that it contains the description of the bookmark, either by its search string, its address or by the URL used to show it on Google Maps online.

- *Directions.plist* seems to contain the current status of the *directions* system of the Maps application. After decoding the file with the *plist* utility, it appears to contain the directions saved as a *data* element:

```
<plist version="1.0">
   <dict>
   <key>DirectionsFileVersion</key>
      <integer>2</integer>
   <key>UserStartSearchProto</key>
      <data>
         CAAQABgEIAA1AQAAAEIGTWls
         ...
      </data>
   </dict>
</plist>
```

  The *data* section contains Base64 encoded strings of all the instructions to reach the destination, along with the distance and time between each of them.

- *History.plist* contains the history of recent searches made by the user in the Maps application. The file contains an array of elements. Each element is a dictionary identified by a numeric key *HistoryItemType* which assumes values 0 or 1.

  Type 0 elements represent a searched address, and are stored in clear text with the text used by the query, latitude and longitude values.

```
<dict>
<key>DisplayQuery</key>
<string>Hard rock cafe' firenze</string>
...
<key>HistoryItemType</key>
   <integer>0</integer>
<key>Latitude</key>
   <real>43.777084350585938</real>
...
<key>Location</key>
   <string>Firenze</string>
<key>Longitude</key>
   <real>11.254043579101562</real>
...
<key>Query</key>
   <string>Hard rock cafe' firenze</string>
</dict>
```

  Type 1 items represent a searched direction (from two positions), and are stored as *data* elements in the format seen above.

In the directory *Library/Maps* there is also a PNG image named *MapIcon.png*. This image shows the position of the temporary bookmark placed on the map by the user. This file is used as the profile image when creating a Contact element from a position on the map, and combined with its creation date and time could represent useful information during a forensics analysis.

Figure 13: Example of MapIcon.png from a test device.

Applications preferences The directory *Library/Preferences* contains a number of binary plist files (66 in the device under test). Each file stores preference for a specific core application (or part of it). The application to which each file belongs is easily identified by its name. During our examinations we found some files containing data which could be useful for forensics purposes, listed below.

- *com.apple.LaunchServices.plist* stores information about the *launch services*, i.e. the user installed applications bound to specific file types. The file is structured as an array of dictionaries, each one containing a *LSHandlerContentType* key describing the file type and a *LSHandlerRoleAll* key specifying the corresponding application.

```
<dict>
   <key>LSHandlerContentType</key>
      <string>com.adobe.pdf</string>
   <key>LSHandlerRoleAll</key>
      <string>com.apple.ibooks</string>
</dict>
```

- *com.apple.Maps.plist* stores information about the status of the Maps application. We found some interesting keys, such as the last viewed latitude, longitude and zoom scale, the start and end strings in the route search fields along with the search strings inserted by the user. A brief extract of the decoded file follows, to show the elements cited before.

```
<key>LastViewedLatitude</key>
   <real>45.563133239746094</real>
<key>LastViewedLongitude</key>
   <real>10.232391357421875</real>
<key>LastViewedZoomScale</key>
   <real>16</real>
...
<key>RouteEndString</key>
   <string>Museo di Storia Naturale, Corso Venezia, 55, 20121
Milano</string>
...
<key>RouteStartString</key>
   <string>Posizione attuale</string>
...
<key>SearchString</key>
   <string>Museo di Storia Naturale, Corso Venezia, 55, 20121
Milano,
      Lombardia</string>
```

- *com.apple.MobileBluetooth.devices.plist* stores a list of all the bluetooth devices paired to the host device. The file is structured as a list of key-dictionary couples. The key contains the MAC address of the bluetooth device and the dictionary contains a list of properties of the device itself, such as a default name, a complete name and the capabilities (handsfree, handset, and so on).

```
<key>00:12:C8:02:42:9D</key>
```

```
<dict>
      <key>DefaultName</key>
   <string>Headset</string>
      <key>DeviceClass</key>
   <data>
      BAQgAA==
   </data>
   <key>Handsfree</key>
      <true/>
   <key>Headset</key>
      <false/>
   <key>Name</key>
      <string>BT HeadSet</string>
...
</dict>
```

- *com.apple.MobileBluetooth.services.plist* stores a list of all bluetooth services supported by the device, and for each service stores a history of paired devices with the date of the last use, along with other information. The file is structured as a list of key-dictionaries. The key contains the name of the service, and the dictionary stores the saved data. As an example we show the data about the handsfree service of the device under test.

```
<key>HandsfreeService</key>
<dict>
   <key>DeviceHistory</key>
   <dict>
      <key>00:12:C8:02:42:9D</key>
      <date>2010-06-21T04:35:48Z</date>
      <key>00:1A:7D:90:F4:84</key>
      <date>2011-02-11T16:32:37Z</date>
   </dict>
<key>State</key>
   <true/>
</dict>
```

- *com.apple.OTASyncAgent.plist* stores data for the OTA (*over the air*) synchronization agent. The most important information found in this file is the Device ID (UDID), the unique device identifier.

```
<dict>
   <key>DeviceId</key>
      <string>81146e (omitted)</string>
</dict>
```

- *com.apple.accountsettings.plist* stores information about the accounts set on the device, for example the accounts used for synchronization purposes or for managing email or notes. The file is structured as a list of dictionaries. Each dictionary is related to a single account and stores data as key-content pairs. The account is identified in each dictionary by the key *Class*. On the device under test 5 account have been found. We provide a brief description of each.

    - Class: *DeviceLocalAccount*. This seems to be the default account existing on the device. It has a list of *Enabled Dataclasses* (Notes and Bookmarks) and a *Type string* showing the value *On My iPod Touch*.

    - Class: *SMTPAccount*. This is an account for sending emails using a Gmail address via SMTP protocol. It contains, among the others, the key *hostname* (which contains the Gmail host address for SMTP services, *http://smtp.gmail.com*) and *Username*.

    - Class: *SMTPAccount*. This is an account similar to the one described before but bound to a Mobile.me account.
    - Class: *GmailAccount*. This file seems to be related to the main Gmail account used to receive email in the device. It contains, among the others, the key *AccountPath* which points to the path where the email files are stored on the host computer (under the Mac Os X directory *~/Library/Mail/*) and the full Gmail username.
    - Class: *LocalAccount*. This file represents the host computer local account, used to store elements not included in the previous one. In the device under test these elements are the Notes and the Outbox folder. This element contains, among the others, the path where the files are stored on the host computer.

- *com.apple.locationd.plist* stores a list of applications allowed to access the location (GPS) capabilities of the device.
- *com.apple.mobilemail.plist* stores settings related to the Mail application. Among other settings, we find the key *SignatureKey*, which contains the default signature appended to every email sent from the device.
- *com.apple.mobilephone.plist* stores data about the telephone application. The most interesting keys found in this file are
    - *AddressBookLastDialedUid*, which stores the user id (as seen in the Contacts database) of the recipient of the last call made by selecting a name from the Contacts application.
    - *DialerSavedNumber*, which stores the last number dialed on the phone.
    - *RecentsLastViewedDate*, which stores the timestamp of the last time the recent calls have been shown to the user.

  An extract from the file is shown as an example.

```
<dict>
   <key>AddressBookLastDialedUid</key>
      <integer>103</integer>
   <key>DialerNumberChanged</key>
      <integer>0</integer>
   <key>DialerSavedNumber</key>
      <string>030 (omitted)</string>
   <key>PhoneAppLastViewType</key>
      <integer>3</integer>
   <key>RecentsLastViewedDate</key>
      <real>320143488.79290301</real>
...
```

- *com.apple.mobilephone.speeddial.plist* stores data about the speed dial function of the mobile phone (the bookmarked numbers for faster dialing). The file is structured as an array of dictionaries, each one about a single bookmark. For each entry the device stores data like the name, the number and the id of the recipient in the address book. An entry is shown below as an example.

```
<dict>
   <key>ABDatabaseUUID</key>
      <string>D67F0904-0820-41BB-AC75-34188064B128</string>
   <key>ABIdentifier</key>
      <integer>0</integer>
```

```
   <key>ABUid</key>
      <integer>160</integer>
   <key>EntryType</key>
      <integer>0</integer>
   <key>Label</key>
      <string>_$!&lt;Home&gt;!$_</string>
   <key>Name</key>
      <string>Casa</string>
   <key>Property</key>
      <integer>3</integer>
   <key>Value</key>
      <string>(omitted)</string>
</dict>
```

The shown entry identifies a speed dial number (omitted, stored in the key *Value*) named "Casa" (key *Name*). It is a phone number (shown by the value 3 in the key *Property*) and is bound to record 160 (key *ABUid*) of the address book.

- *com.apple.mobiletimer.plist* stores user configuration about the two functions of the Clock application: the mobile timer and the world clock.

  For the mobile timer, the file stores an array of dictionaries (under the key *Alarms*), each one representing a single alarm. Among the other pieces of information, the most interesting keys are the alarm time (keys *hour* and *minute*), the title (key *title*) and the last modified time (key *lastModified*). If the alarm is set to start on a specific day of the week, the day is stored in the key *daySetting*. An example of an entry is shown below.

```
<dict>
   <key>alarmId</key>
      <string>2441C01B-1CEC-495D-83B5-C468B5E4DD34</string>
   <key>allowsSnooze</key>
      <true/>
   <key>daySetting</key>
      <integer>0</integer>
   <key>hour</key>
      <integer>8</integer>
   <key>lastModified</key>
      <date>2011-02-23T00:48:03Z</date>
   <key>minute</key>
      <integer>0</integer>
   ...
   <key>title</key>
      <string>Sveglia</string>
</dict>
```

  For the world clock the file stores an array of dictionaries (under the key *cities*), one for each shown clock. Each clock is bound to a specific city, so the data structure stores data like the country name, latitude and longitude, and the timezone.

- *com.apple.preferences.datetime.plist* stores data about the timezone of the device.

- *com.apple.springboard.plist* stores settings regarding the user interface. The most interesting element in this file is the key *SBRecentDisplays*, linked to an array of strings. The strings are the

names of the last opened applications in inverse chronological order, as they are shown in the device's springboard (task manager).

```
<array>
   <string>com.apple.mobilephone</string>
   <string>com.apple.Preferences</string>
   <string>com.apple.mobilemail</string>
   <string>com.linkedin.LinkedIn</string>
   <string>com.facebook.Facebook</string>
```

- *com.apple.stocks.plist* stores the configuration of the Stocks application.
- *com.apple.weather.plist* stores the configuration of the Weather application.
- *com.apple.youtube.plist* stores information about the YouTube application. The data which can be retrieved from this file are the bookmarks and the history of the last seen videos. For each video it stores the eleven character code with which the videos are identified in the YouTube system.

```
<key>Bookmarks</key>
   <array>
      <string>O4ce74q2zqMY</string>
      <string>t9WTviVKbFc</string>
      <string>Kq6_-fvtV4I</string>
   </array>
<key>History</key>
   <array>
      <string>grjIOaE4-aA</string>
      <string>qDgc3fXtiho</string>
...
```

The easiest way to discover the video linked to a code is to insert it in the YouTube URL:
http://www.youtube.com/watch? v=XXXXXXXXXXX
where the Xs represent the code.

Mobile SMS

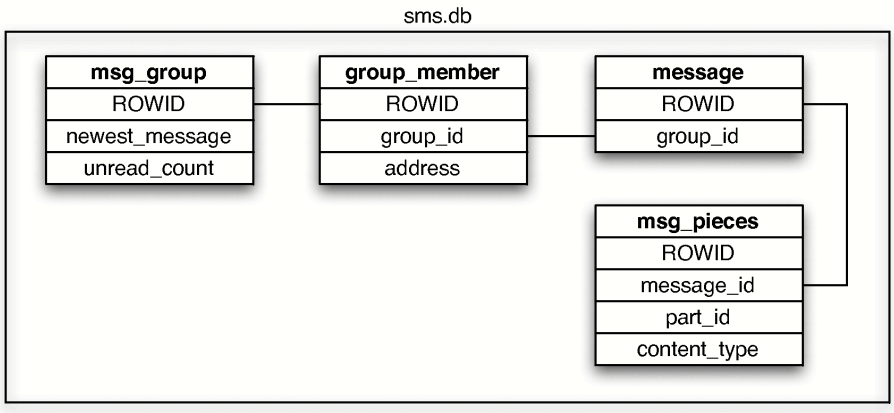


Figure 14: Structure of sms database file.

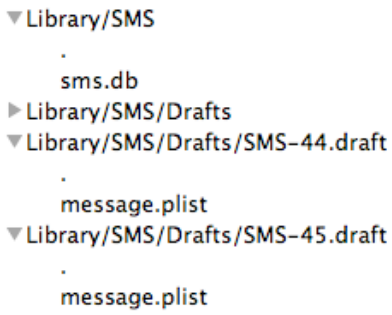


Figure 15: Contents of SMS backup directory.

The directory *Library/SMS* stores the SMS messages in the device. The main storage area is the file *sms.db*, which is a SQLite database with a structure similar to the one shown in figure 14. It is important to know that iOS stores SMS messages as "conversations", i.e. threads of messages sent to and received from a single phone number shown in chronological order. For each group of messages there is a single entry in the *msg_group* table, which also shows the number of unread messages in the thread and the ID of the most recent message. Each element in the previously mentioned table is linked to an element of the specular table *group_member*, which stores the address, i.e. the phone number of the recipient.

Each SMS is a single record in the *message* table. Each record holds the text of the message, the number of the sender/receiver, the id of the message group (as shown in table *msg_group*), a *flags* field which represents whether the message was sent or received (values 3 or 2 respectively), the read/unread flag and the timestamp.

The device stores also drafts, which are text strings written in the text area of the SMS application but not yet sent. These strings are stored in subdirectories of the directory *Library/SMS/Drafts*. Each subdirectory is named after the ID of the group it refers to (see figure 15) and contains a single binary plist file named *message.plist*. This file contains the text, as shown in the example below.

```
<dict>
   <key>markupString</key>
      <string>Testo di prova&amp;nbsp;&lt;br&gt;</string>
   <key>resources</key>
      <array/>
   <key>textString</key>
      <string>Testo di prova</string>
</dict>
```

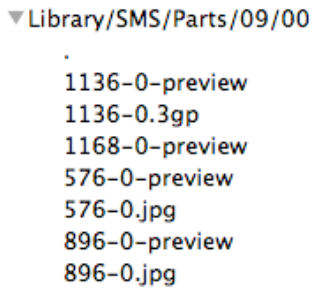


Figure 16: Contents of SMS parts backup directory.

Multimedia messages (MMS) are stored in the same way as regular messages, but without the text. The contents of the messages (which could be multimedia elements and/or text) are stored as separate records in the table *msg_pieces*. Each record of this table contains, among the other data, the content type (image or plain text), the content itself and the ID of the message it belongs to. If the content is an image, it can be contained in the record or be referenced by file name on the multimedia directory of the device. If the referenced content is not stored in the table then it is stored under the *Media* domain, in the directory *Library/SMS/parts*. Each subdirectory contains one or more multimedia elements, often in pairs: the element and its preview (the latter has the same name of the file it references, but without extension and with suffix "-preview"). The name of each element contains the ID of the message it belongs to and an ordinal value.

As an example we show the SQL queries used by the iPBA software to browse through the messages (organized by thread).

To list the threads in the device:

```
SELECT DISTINCT(msg_group.rowid), address FROM msg_group INNER JOIN
group_member
ON msg_group.rowid = group_member.group_id
```

To list the messages in a thread (identified by its ROWID field):

```
SELECT text, date, flags, message.ROWID FROM message INNER JOIN
msg_group
ON msg_group.rowid = message.group_id WHERE msg_group.rowid = <ROWID>
ORDER BY date
```

To list external elements linked by a message (identified by its ROWID):

```
SELECT part_id, content_type, content_loc FROM msg_pieces
WHERE message_id = <ROWID> ORDER BY part_id
```

4.3 Keychain domain

The *Keychain* is a centralized, system-wide storage where iOS applications can store information they consider sensitive. Typically, such information includes passwords, encryption keys and certificates. Data in keychain is always encrypted.

When a user backs up iPhone data in an unencrypted form, the keychain data is backed up but the sensitive data remain encrypted as it was in the device filesystem. The keychain password is a unique device key, which is deemed impossible to reach from outside the device itself. Therefore, passwords and other secrets stored in the keychain on the iPhone cannot be used by someone who gains access to an iPhone backup [6].

This form of backup has a setback: it is not possible to restore the backup onto another device, because it would not know the key used to encrypt the keychain. To address this issue, Apple changed the way keychain backup works in iOS 4. Now, when creating an encrypted backup (the user has set up a password to protect backup) then keychain data is re-encrypted using an encryption key derived from backup password and thus can be restored on another device (by providing the backup password). If the backup password has not been set, then everything works like before iOS 4. Keychain encrypted with device key is included in the backup [7]. We'll explain later why this could be interesting on a forensics perspective.

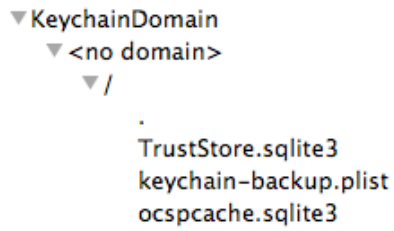


Figure : Contents of Keychain backup domain.

The *Keychain domain* contains files related to security and encryption systems on the device. The most interesting file in this domain is the *keychain-backup.plist*, which stores a copy of the keychain of the device, containing all the stored passwords and certificates. The file is a binary plist file which, after being converted into a plain text plist file with the Mac Os X utility *plutil*, shows a structure composed by an array of dict entries like this one:

```
<dict>
   <key>acct</key>
   <string>Brainld</string>
   <key>agrp</key>
   <string>apple</string>
   <key>pdmn</key>
   <string>ck</string>
   <key>svce</key>
   <string>AirPort</string>
   <key>v_Data</key>
   <data>
      AAAA...
   </data>
   <key>v_PersistentRef</key>
   <data>
      Z2NucAAAAAAAAACP
   </data>
</dict>
```

The one shown in the previous example is the entry extracted from a test device containing the password to access a WiFi network named *Brainld*. The password is stored (in encrypted format) in the first data tag.

Keychain encryption As stated before, the keychain data is always stored in encrypted format, even for unencrypted backups. As we have been able to understand by searching the Internet [8], it appears that the algorithm used to encrypt data is AES with a 256 bits key length. In unencrypted backups (as well as in the device filesystem) the sensitive information is encrypted with a key stored inside the device which can't be accessed from outside. This leads to the conclusion that trying to break this form of protection is not a feasible way.

In encrypted backups, instead, the sensitive data in the keychain has to be re-encrypted using a key derived from the encryption password provided by the user. The symmetric AES key is obtained from this password by iterating on it 10.000 times the PBKDF2 algorithm with an unknown salt. This is a standard practice for key strengthening user supplied passwords for AES, because the process makes brute force attacks more expensive in terms of computational power and time.

Of a forensics perspective knowing that the keychain is encrypted only by a key derived from the user password means that by knowing the details of the encrypting algorithm along with the password itself it

is possible to decrypt the keychain and acquire all the passwords stored in it. Commercial softwares (like *iPhone Password Breaker* from *Elcomsoft*) are reported to be able to accomplish this result.

4.4 Media domain

The Media domain is where all the multimedia information of the device is stored. In this domain we find, for example, the directory *Library/SMS*, which, as stated before, stores multimedia elements (photos and videos) from the MMS archive.

Except for the SMS data, all the Media domain is stored in the device under the directory *Media*. The simplified structure of the Media folder is shown in figure 18. In this section we will describe all the data found in this domain which is deemed interesting for forensics purposes.

The folder *Media/DCIM/100APPLE* stores the unmodified files for the elements in the device's multimedia library (images, audio recordings and videos). The files are all stored in the form *IMG_XXXX.EXT*, where *XXX* is a consecutive number attributed to files during creation and *EXT* is the file type extension. In the device under test we found JPG, PNG image files and MOV, MP4 videos or audio recordings.

Figure 18: Contents of Media domain folder.

Note that JPG images can store EXIF data. EXIF data are pieces of information stored in the file header which can contain various elements about the image and the device used to create it. We found that the amount of EXIF data varies with the application used to take the photo: for example, images taken by the Photo application of the iPhone store the largest amount of data, while photos taken by other applications (such as the Facebook app) store fewer elements. In the following listing we show, as an example, the amount of data found in a photo taken by the iPhone Camera application, read by a Python script.

```
JPG EXIF tags:
Tag: ExifVersion, value: 0221
Tag: DateTimeOriginal, value: 2010:05:01 16:09:04
Tag: DateTimeDigitized, value: 2010:05:01 16:09:04
Tag: Make, value: Apple
Tag: Model, value: iPhone 3GS
Tag: Orientation, value: 6
Tag: GPSInfo, value: --omitted--
Tag: Software, value: 3.1.2
Tag: DateTime, value: 2010:05:01 16:09:04
Tag: ExifImageWidth, value: 2048
Tag: ExifImageHeight, value: 1536
Tag: ExifOffset, value: 206
...
```

For forensics purposes, the most important tags are *DateTimeOriginal*, which stores the date and time when the photo was taken, and *GPSInfo*, which stores the geographic position where the picture was taken according to the iPhone GPS unit. The GPS data is stored as a list of key-value pairs, as depicted in EXIF 2.2 standard [9]. The keys usually found in pictures in the device under test are:

- 1 (*GPSLatitudeRef*): Indicates whether the latitude is north ('N') or south ('S').
- 2 (*GPSLatitude*): Indicates the latitude. The latitude is expressed as three rational values giving the degrees, minutes, and seconds, respectively.
- 3 (*GPSLongitudeRef*): Indicates whether the latitude is west ('W') or east ('E').
- 4 (*GPSLongitude*): Indicates the longitude. The longitude is expressed as three rational values giving the degrees, minutes, and seconds, respectively.
- 7 (*GPSTimeStamp*): Indicates the time as UTC. TimeStamp is expressed as three rational values giving the hour, minute, and second.
- 16 (*GPSImgDirectionRef*): Indicates the reference for giving the direction of the image when it is captured. 'T' denotes true direction and 'M' is magnetic direction.
- 17 (*GPSImgDirection*): Indicates the direction of the image when it was captured. The range of values is from 0.00 to 359.99.

As an example we show a GPS tag extracted from the EXIF data of an image found in the device under test.

```
Tag: GPSInfo, value: {1: 'N', 2: ((45, 1), (3010, 100), (0, 1)),
3: 'E', 4: ((9, 1), (1028, 100), (0, 1)),
7: ((13, 1), (27, 1), (89, 100))}
```

This data shows that the photo was taken at latitude 45 30.10' 0" N, 9 10.28' 0" E at 13:27 UTC.

The folder *Media/PhotoData* contains a plist file (storing the date of the last modification of photo databases) and two SQLite databases containing information about the photos and videos stored in the device. The simplified structure of the databases is shown in figure 19.

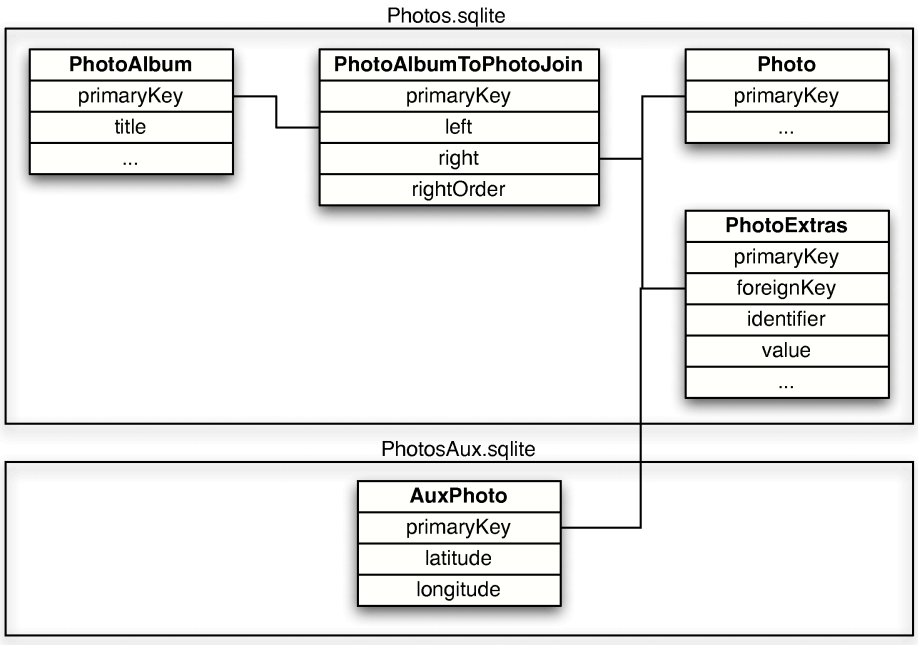


Figure 19: Structure of photos database file.

As can be seen, the main table is the one named *Photo* (in *Photos.sqlite*), where there is a record for each photo or video. In this record we found, among other information:

- Width and height in pixels.
- Thumbnail index: the position in the .ithmb files (see appendix B) where we can find the thumbnails of the photo or video.
- Directory and filename: where the original file is stored (usually in *Media/DCIM/100APPLE*).
- Capture time: timestamp of when the photo (or video) was captured.
- Duration: the duration of the video (0 for image records).
- Orientation: 6 for portrait or 1 for landscape orientation.

The records in table *PhotoExtras* are linked to the *Photo* records described before and seem to contain additional information. The records in table *AuxPhoto* (in file *PhotosAux.sqlite*) store latitude and longitude for each photo.

The folder *Media/PhotoData/100APPLE* contains preview data for the video files seen in folder *Media/DCIM/100APPLE*. For each video file this folder contains two or three files with the same name as the original file but different extension:

- *IMG_XXXX.JPG* is an image with a preview of the first frame of the video.
- *IMG_XXXX.THM* is also a JPG image with a small, square preview of the first frame of the video.
- *IMG_XXXX.THP* is a data file which exists only for MOV video files and stores preview frames of the video. The frames are stored (after a 16 bit header) as 22x29 pixel raw images (with no padding between each frame), in the same format used to store thumbnails (see appendix B).

The folder *Media/PhotoData/Thumbnails* contains three files:

- *thumbnailConfiguration*
- *120x120.ithmb*
- *79x79.ithmb*

The first one is a plain text plist file containing version numbers for the thumbnail management system. The other two are raw graphic files storing the thumbnails for all the images in the multimedia library of the device. These files are further described in appendix B.

The folder *Media/PhotoData/Videos* stores in its subdirectory structure the BTH files for the video in the multimedia library. The files have the same name of the video they refer to but with BTH extension (*IMG_XXXX.BTH*). The BTH files contain a preview of some frames of the video. Their structure is described in appendix C.

The folder *Media/Recordings* stores data for the Audio Recorder application. In this folder we found:

- The recorded audio files in M4A format.
- A plist file named *CustomLabels.plist*.
- A SQLite database file named *Recordings.db*.

In the SQLite database the only interesting table is *ZRECORDING*, which stores a record for each recording. For each recording the table stores the duration (in seconds), the timestamp, the complete path of the audio file and the index of the label used to name the recording on the device (there are many preset labels). If the label code is 7 it means that the user created a custom label, which is also stored in a field of the record.

```
Table Record:
...
- ZLABELPRESET : 7
- ZDURATION : 3.07329499722
- ZDATE : 321360293.975
- ZPATH : /var/mobile/Media/Recordings/20110309 114453.m4a
- ZCUSTOMLABEL : My new label
```

The file *CustomLabels.plist* contains a dictionary of key-string pairs, in which the key is the complete name of a recorded audio file (with extension) and the string is the label associated (being it a standard label or a custom made one).

```
<dict>
   <key>20110309 114453.m4a</key>
   <string>My new label</string>
   <key>20110309 114535.m4a</key>
   <string>Podcast</string>
</dict>
```

4.5 Root domain

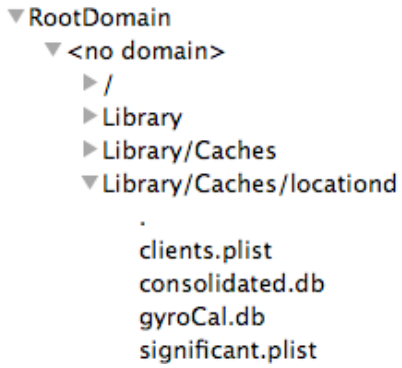


Figure 20: Contents of Root domain folder.

In the device under test the only content of the Root Domain appears to be the directory *Library/Caches/locationd* which, as the name suggests, seems to hold information about the location capabilities of the iPhone. The structure of the domain is shown in figure 20.

The file *clients.plist* is a binary plist file. It contains a list of key-dictionary pairs. Each key holds the name of an application which has requested access to the location capabilities, and the corresponding dictionary contains information like:

- Whether the application has been authorized to access the location data.
- The name of the main executable of the application.
- The timestamps of when the location acquisition has been started and stopped the last time by the application.

An example of an entry of the file is shown.

```
<key>com.apple.Maps</key>
<dict>
   <key>Authorized</key>
   <true/>
   <key>BundleId</key>
   <string>com.apple.Maps</string>
   <key>Executable</key>
   <string>/Applications/Maps~iphone.app/Maps~iphone</string>
   <key>LocationTimeStopped</key>
   <real>321223417.15129601955413818359375</real>
   <key>PromptedSettings</key>
   <false/>
</dict>
```

The file *significant.plist* is an empty binary plist, and its use is not known. The same applies to the file *gyroCal.db*, a SQLite database which contains just an empty table *GyroCalibration*. According to its name, we think it is used for the calibration of the gyroscope, but it contains no data so it is unknown whether it is used and in which way.

The file which seems most interesting on a forensics perspective is *consolidated.db*. It is a SQLite database file which seems to contain the data used by the device assisted GPS. The assisted GPS is a system to achieve an approximate localization of the device by using data from mobile cells and wireless

networks found nearby, and it is used while the main GPS system isn't ready to give the exact position. To achieve this result the device needs to store a cache of known cells and WiFi networks with their approximate geographical position.

The most interesting tables in *consolidated.db* appear to be *WifiLocation* and *CellLocation*. The table *CellLocation* seems to store a cache of Base Stations (one for each record), identified by their MCC[7], MNC[8], LAC[9] and CI[10]. As an example we show a sample record from the table:

```
- MCC : 222
- MNC : 1
- LAC : 55231
- CI : 18051
- Timestamp : 299055421.65
- Latitude : 45.28178596
- Longitude : 11.49680888
- HorizontalAccuracy : 2974.0
- Altitude : 0.0
- VerticalAccuracy : -1.0
- Speed : -1.0
- Course : -1.0
- Confidence : 70
```

Each record stores:

- A single cell identification: MCC, MNC, LAC and CI.
- A timestamp.
- A geographic position (latitude, longitude, altitude).
- Vertical and Horizontal accuracy.

While it is not known exactly how the system works, it has been verified that for a single value of timestamp (i.e. for a single instant) many records are recorded, all belonging to a limited area. It has also been verified that the mobile phone was effectively located in the area among the base stations found at the time recorded by the timestamp. So we can reasonably assume that the device stores, at the time specified by the timestamp, data about all the base stations the device "knows" and their approximate geographical position (probably this data is temporarily stored elsewhere as it is harvested, maybe in the now empty table *CellLocationHarvest*, and then written to this table). This information is surely invaluable for a forensics examiner, because it can be used to prove that a seized device was in a known limited area at a certain time. In the device under test we have been able to recover even one year old positioning data. The locations can be extracted from the table by a simple SELECT statement and then used to create a KML file which in turn will be shown by a geographical application such as Google Maps. An example of an extraction is shown in the following query and figure 21, in which a bold line connects all the cells showing the approximate location of the device at the specified time.

```
SELECT Longitude, Latitude
  FROM CellLocation
```

[7] MCC: Mobile Country Code. The code identifies the country of the operator.
[8] MNC: Mobile Network Code. The code identifies the mobile provider.
[9] LAC: Location Area Code. The code identifies a set of base stations grouped together to optimise signalling.
[10] CI: Cell Identity. The code identifies a single Base Station.

```
WHERE Latitude <> 0
and Timestamp = 312276699.97992
and HorizontalAccuracy < 1000
LIMIT 10000;
```

The table *WifiLocation* seems to work the same way. Each record of this table stores data about a WiFi device (identified by its MAC address) and its location:

- A single WiFi device identification: MAC address.
- A timestamp.
- A geographic position (latitude, longitude, altitude).
- Vertical and Horizontal accuracy.

Like the previous table we can reasonably assume that the device stores these elements while they are harvested, and at a certain time (recorded by the timestamp) these are written to the *WifiLocation* table. We verified that the location data we have been able to recover by these assumptions proved correct. As an example, we show the SQL query used to extract positioning data at a certain time, and the data as plotted on Google Earth (see figure 21).

```
SELECT Longitude, Latitude
 FROM wifilocation
 WHERE timestamp = "299089059.116161"
 LIMIT 100;
```

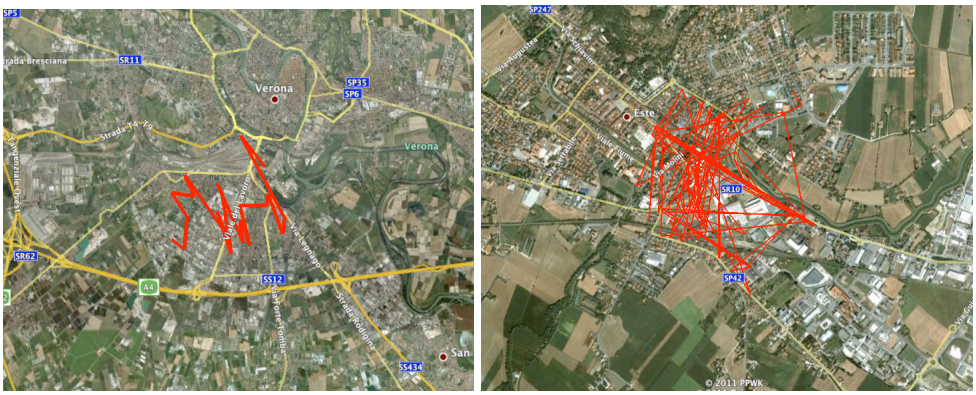


Figure 21: Cell and WiFi location, each for a single timestamp (images from Google Earth).

4.6 System Preferences Domain

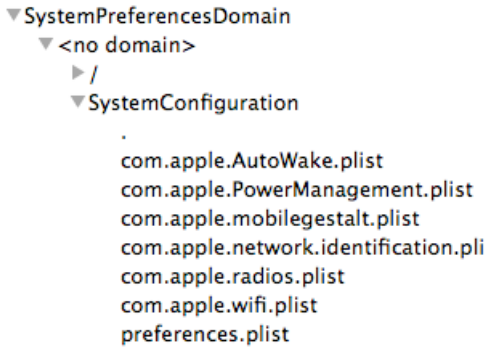


Figure 22: Structure of the System Preferences domain folder.

The domain *System Preferences* appears to contain only the directory *SystemConfiguration*, which stores some binary plist files holding information about the configuration of the core components of the iOS system. The files in the System Preferences Domain of the device under test are depicted in figure 22.

The files found in the device are:

- *com.apple.AutoWake.plist* appears to contain a chronological list of the future events which will need to automatically wake the device. For each event it indicates the event type, the process which scheduled the event and the time. As an example we show the event description of the waking of the device for the automatic fetch of emails by the Mail client.

```
<dict>
   <key>eventtype</key>
   <string>wake</string>
   <key>scheduledby</key>
   <string>com.apple.persistentconnection[MobileMail,52,0x6038670,
      MailAutoFetch]</string>
   <key>time</key>
   <date>2011-03-09T11:45:45.000000Z</date>
</dict>
```

- *com.apple.PowerManagement.plist* seems to store the date of the last sleep event along with an unique identifier.

- *com.apple.mobilegestalt.plist* seems to store only an user assigned device name ("iPhone" in the device under test).

- *com.apple.network.identification.plist* seems to store data about the networking devices the iPhone has been in contact with. It appears to be composed of a list of "identifiers" (the IP address of a router and the corresponding hardware address, or the name of the interface for a cellular WAN), each one of them associated to a timestamp and an array of "services".

```
<key>Identifier</key>
<string>IPv4.Router=192.168.10.1;
   IPv4.RouterHardwareAddress=00:26:5a:fe:d2:7e</string>
<key>Services</key>
<array>
...services...
</array>
<key>Signature</key>
<string>IPv4.Router=192.168.10.1;
```

```
   IPv4.RouterHardwareAddress=00:26:5a:fe:d2:7e</string>
<key>Timestamp</key>
<date>2011-03-08T22:00:57.884106Z</date>
```

Each service is in turn a dictionary containing DNS and IPv4 data, as shown below.

```
<dict>
   <key>DNS</key>
   <dict>
      <key>ServerAddresses</key>
      <array>
         <string>62.13.169.92</string>
         <string>62.13.169.93</string>
      </array>
   </dict>
   <key>IPv4</key>
   <dict>
      <key>Addresses</key>
      <array>
         <string>1.219.1.44</string>
      </array>
      <key>InterfaceName</key>
      <string>pdp_ip0</string>
      <key>Router</key>
      <string>1.219.1.44</string>
      <key>SubnetMasks</key>
      <array>
         <string>255.255.255.255</string>
      </array>
   </dict>
   <key>ServiceID</key>
   <string>3EE6AED4-4969-441A-A2C9-3137439DAC0C</string>
</dict>
```

From the example above we could reasonably assume that the device has been attached to a cellular network by its hardware interface named `pdp_ip0`. In the listing we could find the addresses of the DNS servers, the address of the router, the address assigned to the mobile phone and the subnet mask of the network. These elements could be forensically useful by letting the examiner know the device has been effectively attached to a network, along with its IP address.

- *com.apple.radios.plist* seems to store only whether the device is in airplane mode or not. Being in airplane mode means that all the radio components in the device (GSM/UMTS, Bluetooth, WIFi) have been disabled.

- *com.apple.wifi.plist* stores a list of known WiFi networks. For each network the file contains, among the others, BSSID, SSID (name of the network), security mode (WEP/WPA), strength of the signal, channels used and the timestamps of the last manual join and autojoin. In the following listing, as an example, only the most forensically interesting elements are shown for a single known network.

```
...
<key>BSSID</key>
   <string>0:26:5a:fe:d2:7e</string>
...
<key>CHANNEL</key>
   <integer>1</integer>
```

```
...
<key>RSSI</key>
   <integer>-66</integer>
<key>SSID</key>
   <data>
      YnJhaW5sZA==
   </data>
<key>SSID_STR</key>
   <string>brainld</string>
...
<key>SecurityMode</key>
   <string>WPA Personal</string>
<key>Strength</key>
   <real>0.73829400539398193359375</real>
...
<key>lastAutoJoined</key>
   <date>2011-03-09T10:44:44.880916Z</date>
<key>lastJoined</key>
   <date>2010-04-15T17:37:22.000000Z</date>
...
```

4.7 Wireless Domain

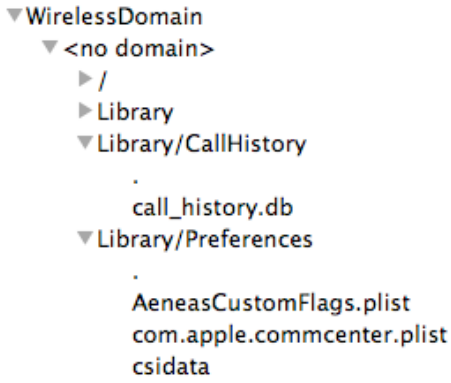


Figure 23: Structure of the Wireless Domain domain folder.

The *Wireless Domain* holds data related to the mobile phone part of the device. The structure of the domain is shown in figure 23.

The domain is made up of two directories. The first, *Library/CallHistory*, contains just one file, a SQLite database named *call_history.db*. As the name suggests, the database stores in a table called *call* one record for each of the last 100 phone calls the device received, missed or made. The structure of each record is depicted below:

```
Table Fields:
- 0 "ROWID" (INTEGER)
- 1 "address" (TEXT)
- 2 "date" (INTEGER)
- 3 "duration" (INTEGER)
- 4 "flags" (INTEGER)
- 5 "id" (INTEGER)
```

```
- 6 "name" (TEXT)
- 7 "country_code" (TEXT)
```

The field *address* stores the number of the other phone involved in the call. The field *date* stores the timestamp of the call (in Unix Epoch time format, i.e. the number of seconds elapsed since January 1st, 1970). The field *duration* is the duration of the call in seconds. The value is zero for unanswered calls. The field *flags* is an integer value used to distinguish between outgoing (5) and incoming (4) calls. It appears there could be other values of this field: in the device under test we found a value "1507333" corresponding to a couple of outgoing calls failed due to network problems. We didn't find any other values. The field *country code* appears to be, as the name suggests, the code of the country the call was originated from. In the device under test the value 222 (Italy) was found for almost every call, except for some calls made/received from the device when it was located in Paris (France) to Italian numbers, marked with a country code 208 (France). This value can be useful for forensics purposes by enabling an examiner to know in which country the phone was when it made/received calls. The field *name* is always empty and does not seem to be used. The field *id* appears to contain the ID of the recipient of the call as the index of a record in the Contacts table. The value is used only for outgoing calls to numbers from the Contacts (all the incoming calls have this value set to -1).

The file *call_history.db* contains another table called *data* which appears to store logs for UMTS data connection of the device. The structure of the records in the table is:

```
Table Fields:
- 0 "ROWID" (INTEGER)
- 1 "pdp_ip" (INTEGER)
- 2 "bytes_rcvd" (REAL)
- 3 "bytes_sent" (REAL)
- 4 "bytes_last_rcvd" (REAL)
- 5 "bytes_last_sent" (REAL)
- 6 "bytes_lifetime_rcvd" (REAL)
- 7 "bytes_lifetime_sent" (REAL)
```

In the device under test we found that the file has four records, the last three of them with all the "bytes_" fields set to zero. We can assume that each record was designed to store data for a single network interface (as it appears to be described by the field "pdp_ip"). In fact, as seen in previous logs, the iPhone appears to use just the interface *pdp_ip0*, which is why the record with pdp_ip = 0 is the only populated one.

In the only populated record, the fields *bytes_rcvd* and *bytes_sent* are the values of incoming and outgoing data traffic (in kilobytes) from the last reset of the counters. The fields *bytes_lifetime_rcvd* and *bytes_lifetime_sent* are the same values as stated before but can never be zeroed by the user and so represent the total amount of data traffic of the device during its entire lifetime. The fields *bytes_last_rcvd* and *bytes_last_sent* appear to be the data traffic (always in kilobytes) for the last connection.

The file *call_history.db* has another table named *_SqliteDatabaseProperties* in which each field is made up by a pair key-value and appears to store values related to the timers measuring the length of calls made by the device. The reason these values are stored in a proprietary table (in a different way from the data transfer logs) seems to be that the values in this table are updated by database triggers whenever a new record is added to the *call* table. The table contains records for the data transfers counts, too, but those values are all zero in the device under test and do not seem to be used. The most notable values found in this table are:

- *call_history_limit*: maximum number of call records stored in the *call* table.
- *timer_last*: duration (in seconds) of the last call.

- *timer_incoming* and *timer_outgoing*: duration (in seconds) of incoming and outgoing calls from the last counters reset.
- *timer_all*: sum of the previous values.
- *timer_lifetime*: sum of durations of all incoming and outgoing calls in the lifetime of the device.

5 iPBA - iPhone Backup Analyzer

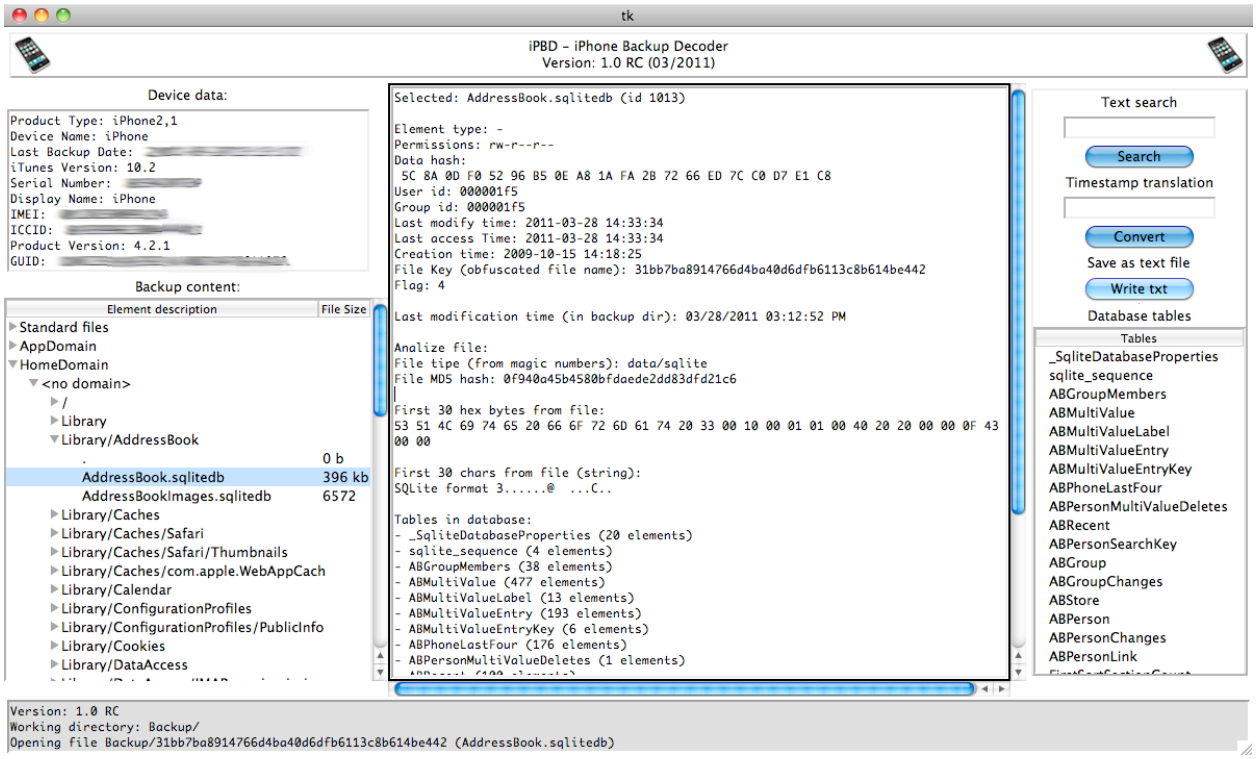


Figure 24: Main user interface of iPBA.

To analyze the backup data in a simple manner we developed a software tool, called *iPBA - iPhone Backup Analyzer*. This tool provides a simple mean to browse through a backup directory and perform a first analysis of each element contained. It is written in Python and Tk, and thus it should be able to run on each platform which supports this scripting language (it has been tested on Mac Os X and Linux).

The software is started by launching the main script file, passing via command line the location of the directory, such as:

```
./main.py -d Backup/
```

Upon startup, the tool locates the index files *Manifest.mbdb* and *Manifest.mbdx* and parses them for all the data related to each single element they describe. This data is then stored in a SQLite database built in RAM (one record of a table for each element). Another table stores the objects properties, as each object could have an arbitrary number of properties.

After parsing the index files, the user interface is built (see figure 24). In the left column there is a tree to show the elements in an ordered manner: first by their domain, then by their subdomain (only for the Applications Domain), then by their path and at last by their filename. In the upper part of the left column

are shown properties about the backup, parsed from the *Info.plist* file, such as the version of iOS on the device, the date of the backup and unique identifiers of the iPhone (ICCID, IMEI).

When the user clicks on a filename in the tree, the software tries to analyze it and provides the output in the main text area. The data collected from the index files are written first:

- File type (file, directory, symbolic link).
- Unix permissions.
- Data hash (if stored in the index files).
- User and Group ID.
- Last modified time, last access time, creation time.
- File key (the name of the file in the backup directory).
- Flag.
- File properties (if any).

Then more data are provided by analyzing the file itself:

- Last modification time of the file in the backup directory.
- File type (by magic numbers).
- MD5 hash.
- First 30 bytes of the file in hexadecimal format.
- First 30 bytes converted to ASCII characters.

Finally, a deeper analysis is conducted based on the type of the file:

- If the file contains ASCII text, the whole content of the file is shown in the main text area. The same applies when we are analyzing plain text plist files.
- If the file is a binary plist, it is automatically converted to plain text plist by an external utility written in Perl (on a temporary file). The temporary file is then shown in the main text area.
- If the file contains data, then the file content is displayed as an hexadecimal dump. The utility displays on each line of the main text area the input offset, followed by sixteen space-separated bytes of data, followed by the same bytes converted to their ASCII counterpart.
- If the file is a recognized image (such as PNG or JPG), the image itself is shown in the main text area. If the image is a JPG, then the software displays a list of all the EXIF data it contains.
- If the file is a SQLite database, then the software displays a list of all the tables in the file, each with the number of records it contains.

When a SQLite file is selected, in the right column the software displays a list of the tables in the file. When the user clicks on a table, the tool dumps the content of the table (preceded by a description of the table structure) in the main text area.

The software is still in active development, and has been fitted with a couple of experimental functions which will be extended in future releases, such as a search function, a timestamp converter (from absolute time format) and an option to export the text in the main text area to an external file. Future versions of iPBA will continue to improve the analysis of each object, by providing additional built-in tools to show additional data more related to each file type (for example a way to decode thumbnail files,

which as shown above are built in a non standard format). Other improvements will lead to provide personalized functions to decode and present important data for which the structure has been recognized (for example, we could provide a function to exploit the structure of the SMS database to show the conversations in a iPhone-like style). And to make the software useful in a forensics examination it will need reporting functions with integrity check of the objects.

5.1 Practical informations

iPBA *iPhone Backup Analyzer* has been released as open source software under the MIT license. Further informations about the software itself, along with links to download the code and screenshots are provided at the address: `http://ipbackupanalyzer.com`

We chose to distribute the software as open source mainly to provide a common platform for the analysis of iOS backup data; all the interested people are encouraged to participate, by contributing with new code, fixing bugs or by just testing the software and making suggestions.

6 Conclusions

This study explored the forensics examination of the content of an iPhone device by exploiting the backup data acquired by the iTunes software. The examination process tried to make a comprehensive identification of all the objects found among the thousands of files contained in the backup directory. During this study an application has been developed to make the process faster and simpler.

During this research we have been able to locate a significant number of pieces of data which constitute the first objectives of a forensics analysis of a smartphone (such as contacts, sms data, browser data and so on) along with the objectives required by the analysis of a complex device like the iPhone (applications data, notes, audio memos and so on). We have been able to uncover hundreds of elements and provide a brief description of each, along with hints about their usefulness in a forensics analysis and instructions to build tools to further analyze them.

iPhone forensics is an evolving field, first of all because of the continuous changes in the structure of the operating system which leads to modifications in how the data is stored and formatted. The structures we described in this research will be probably subjected to modifications in the following versions, so the first goal of the mobile forensics community should be to keep an open eye on future releases of iOS to uncover these modifications and keep the knowledge of iOS up to date.

A Timestamps in iOS

The iOS system uses a mix of Unix timestamp and Absolute Time format to store date and time values.

The Unix time (or POSIX time) is a format in which the time is defined by the number of seconds elapsed since midnight Coordinated Universal Time (UTC) since January 1, 1970 (known as Unix Epoch), not counting leap seconds.

For example, the date *March 14th, 2010 at 11:12:13* is represented in Unix Time by the number `1268565133`.

The Absolute Time is a format in which the time is defined by the number of seconds elapsed since January 1, 2000 00:00:00 GMT [10].

For example, the date *December 16th, 1999 at 17:54:34* is represented in Absolute Time as `-32940326` (it is less than zero because the date is before the Absolute Time zero).

B Structure of thumbnails files

Each one of the thumbnails files described in section 4.4 is a binary file which contains raw images stored in RGB format in 555 mode. The 555 mode (often referred to as *Highcolor*) is a RGB representation of an image in which each pixel is represented by a 16 bit value. Of these 2 bytes, only the 15 least significant bits are used to render red, green and color values with 5 bits each (as shown in figure 25).

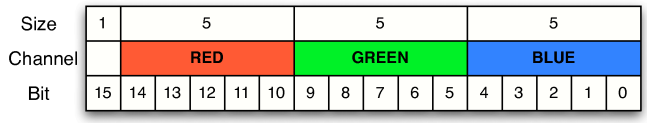


Figure 25: RGB 555 pixel format.

Images are stored in consecutive fixed-length sections. Each section is composed of a number of fixed length rows. At the end of each image there is a fixed length padding. Figure 26 shows the structure of a generic thumbnail file containing images of MxN pixels, with a padding of K 16-bit values between each thumbnail.

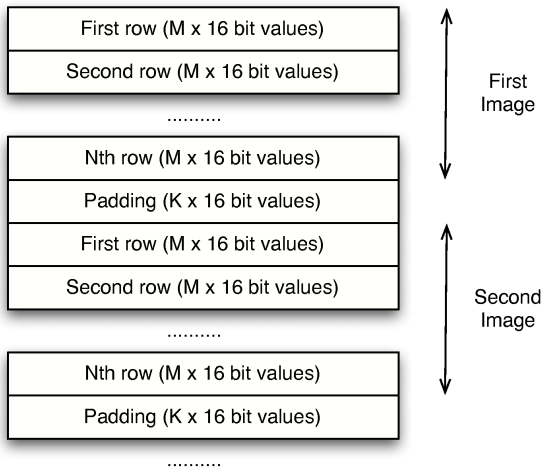


Figure 26: Structure of thumbnails file.

In the device under test we found 2 thumbnail files under the media domain, named:

- 120x120.ithmb
- 79x79.ithmb

Each file contains the thumbnails for all the images in the Image Library of the device. The first file stores 120x120 pixels images with a padding of 14x2 bytes between each. Note that the thumbnails do not fill the available space because they are rectangular, so they have to be trimmed accordingly to their format (portrait or landscape). The unused space in the canvas is filled with color black (`0x0000`). See figure 27.

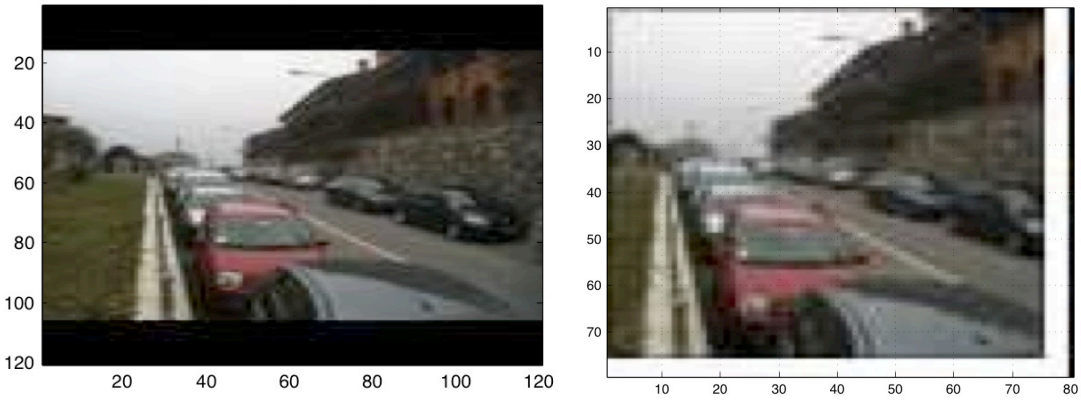


Figure 27: A 120x120 and a 80x80 thumbnail of the same image.

The second file stores 80x79 pixels images (of which only 79x79 are effectively used) with a padding of 14x2 bytes between each. The unused 80th column is filled with color black (`0x0000`). Note that, as seen before, the images usually do not fill the available space. The unused space in the canvas is filled with color white (`0xFFFF`). See figure 27.

A sample Matlab code which can be used to show a thumbnail in the file is shown in listing 1.

```
clear
%-----reads thumbnails file
filename='dfd4021bd3cd3737c86bfc2fa675dd96136a1b1c';
fid = fopen(filename, 'r+');
data = fread(fid, 'uint16');
%-----thumbnail index
i=20;
%-----skip dimension (in uint16 values)
skip = 14;
%-----image dimension
imagerows = 80;
imagecolumns = 79;
framelen = imagerows * imagecolumns + skip;
%-----reads the selected image into a matrix
frame = data(framelen * i + 1 : framelen * (i+1) );
frame = reshape(frame(1:(end-skip), :), imagerows, imagecolumns);
frame = frame';
%-----splits the image in the RGB components (normalized in range [0;1])
rgb(:,:,1) = bitshift(bitand(frame, bin2dec('0111110000000000')), -10) ./ 31;
rgb(:,:,2) = bitshift(bitand(frame, bin2dec('0000001111100000')), -5) ./ 31;
rgb(:,:,3) = bitshift(bitand(frame, bin2dec('0000000000011111')), 0) ./ 31;
%-----shows image
image(rgb);
grid on;
fclose(fid);
```

Listing 1: Example of Matlab code to analyze 79x80 thumbnails file.

The meaning of the 14 2-bytes values of padding between each image is not known. The values change in an apparently random manner between each image. The only values for which we have been able to recognize a meaning are the ones from 9 to 14. Elements 10, 12, 13, 14 are always zero. Elements 9 and 11 appear to be the real width and height of the image in pixels. These values must be used to crop away the borders of the fixed-size thumbnail. For example, in *120x120.ithmb*, images in portrait format are 90x120, while landscape images are 120x90.

One interesting aspect of the thumbnail system is that it can be used to recover deleted images from the device. We verified that after an element has been deleted from the multimedia library, its thumbnail is not immediately removed from the thumbnails files. Instead, the thumbnail data is still present, at least until it is overwritten by a new thumbnail. The only difference found in the thumbnails files after the deletion of an image is that the 14x2 bytes after the thumbnail data have all been zeroed in both files, except for the two values of width and height.

C Structure of BTH files

The files with extension BTH appear to be preview data for video files in the multimedia gallery of the device. The files have the following structure:

- Header: a 36 bytes header, starting with hex values `EA CD C4 DF`. This header contains, at offset 20, the hex values `AB DC CD BA`, already seen as the header of THP files.
- Image: a thumbnail of the first frame of the video with graphics depicting the length of the video itself in minutes and seconds. The thumbnail is a 79x78 pixel image saved as raw bitmap data (the same format used to store thumbnails, already seen in appendix B). See for example figure 28.

Figure 28: A thumbnail extracted from a BTH file.

- Plist data: starting from offset 12518 bytes the file embeds a binary plist, which appears to contain properties of the preview. The only information that appears useful is the length of the video in seconds. The following example shows the duration extracted from the preview file of a video 1 minute and 1 second long.

```
.....
<string>duration</string>
<real>60.600000000000001</real>
.....
```

The bplist data has been extracted and interpreted by using the *plist* and *dd* command line utilities.

```
dd bs=1 skip=12518 < infile > outfile
plutil -convert xml1 -o outfile.plist  outfile
```

References

[1] Mona Bader and Ibrahim Baggili. iPhone 3GS Forensics: Logical analysis using Apple iTunes Backup Utility. *Small Scale Digital Device Forensics Journal*, 4 (1), september 2010.

[2] Understanding file permissions on Unix: a brief tutorial. URL `http://www.dartmouth.edu/ rc/help/faq/permissions.html`. Retrieved February, 2011.

[3] MBDB and MBDX Format. URL `http://code.google.com/p/iphonebackupbrowser/wiki/MbdbMbdxFormat`. Retrieved February, 2011.

[4] SQLite Wikipedia article. URL `http://en.wikipedia.org/wiki/SQLite`. Retrieved February, 2011.

[5] Plist Wikipedia article. URL `http://en.wikipedia.org/wiki/Property\s\do5(l)ist`. Retrieved February, 2011.

[6] Mac Os X Reference Library: Keychain Services Concepts, a. URL `http://developer.apple.com/library/mac/#documentation/Security /Conceptual/keychainServConcepts/02concepts/concepts.html`. Retrieved February, 2011.

[7] Peeking Inside Keychain Secrets, b. URL `http://blog.crackpassword.com/2010/08/peeking-inside-keychain-secrets/`. Blog post retrieved February, 2011.

[8] Cracking Blackberry Backup Passwords. URL `http://blog.crackpassword.com/2010/09/`.

[9] Exchangeable image file format for digital still cameras: Exif version 2.2. Technical report, Japan Electronics and Information Technology Industries Association - Technical Standardization Committee on AV & IT Storage Systems and Equipment, April 2002. Retrieved March, 2011, from *http://exif.org/Exif2-2.PDF*.

[10] Cfdate reference on mac os x developer library. URL `http://developer.apple.com/library/mac/documentation/CoreFoundation/Reference/CFDateRef/Reference/reference.html`. Retrieved on March, 2011.

[11] Steve Whalen. *iPhone Processing*. Forward Discovery, inc. Retrieved February, 2011, from *http://computer-forensics.sans.org/summit-archives/2008/files/iphone-forensics.pdf*.

[12] Jens Heider and Matthias Boll. *Lost iPhone? Lost Password! - Practical Consideration of iOS Device Encryption Security*. Fraunhofer Institute for Secure Information Technology. Retrieved February, 2011, from *http://www.sit.fraunhofer.de/en/Images/sc_iPhone%20Passwords_tcm502-80443.pdf*.