TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Janika Pirel Jasmin Hirvi - 182514IVCM

AN ANALYSIS OF DATABASE MANIPULATION TECHNIQUES IN IOS

Master's thesis

Supervisor: Matthew James Sorell

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Janika Pirel Jasmin Hirvi - 182514IVCM

IOS-I ANDMEBAASIDE MANIPULEERIMISE MEETODITE ANALÜÜS

Magistritöö

Juhendaja: Matthew James Sorell

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Janika Pirel Jasmin Hirvi

15.05.2023

Abstract

This study examines the realm of database manipulation and detection techniques in iOS devices. It highlights the significance of understanding how data is stored, modified, and potentially manipulated within mobile device databases. The paper emphasizes the importance of detecting data manipulation, the complexities involved in recovering truthful data, and the prevalence of jailbroken devices as a challenge in digital forensics. The findings underscore the need for reliable and irrefutable digital evidence in legal proceedings, as the admissibility of evidence can greatly impact the outcome of a case. The research explores various techniques and tools used to detect manipulation traces, ensuring the integrity and completeness of digital evidence.

The study revealed that databases in iOS devices can be manipulated to such an extent that the detection of such manipulations becomes extremely challenging, if not entirely indistinguishable. The concept of corrupting the database like "Ghost messaing", as explored in this study, represents a novel and previously unexplored area of investigation. "Ghost messaging" refers to the phenomenon where messages or data records exist transiently within a mobile device's memory or temporary storage but are not permanently stored in the device's database. Furthermore, the research underscores the necessity of continuous research and development to stay ahead of evolving manipulation techniques employed potentially being employed.

This thesis is written in English and is 58 pages long, including 10 chapters, 33 figures and 3 tables.

Annotatsioon

iOS-i andmebaaside manipuleerimise meetodite analüüs

Antud uuring analüüsib andmebaaside manipuleerimise ja tuvastamise tehnikaid iOS seadmetes. Artikkel rõhutab andmete manipuleerimise tuvastamise tähtsust, tõese andme tuvastamise keerukust ning *jailbroken* seadmete levimusest tulenevat väljakutset digitaalses kohtuekspertiisis. Uurimistöö rõhutab usaldusväärse digitaalse tõendi vajadust õigusprotsessides, kuna andmete tõesus võib oluliselt mõjutada kohtuistungi tulemust. Uurimustöö analüüsib erinevaid tehnikaid ja tööriistu manipulatsioonide tuvastamiseks, tagades digitaalse tõendi terviklikkuse ja täielikkuse. Uurimustöö näitas, et iOS seadmetes saab andmebaase manipuleerida sellisel määral, et selliste manipulatsioonide tuvastamine muutub äärmiselt keeruliseks, kui mitte täiesti eristamatuks.

Uurimustöös analüüsitakse ka "ghost messaging" kontseptsiooni st andmebaaside korrupeerimist, mis esindab varasemalt uurimata valdkonda. "Ghost messaging" all mõistetakse olukorda, kus sõnumid või andmeread eksisteerivad ajutiselt RAM mälus, kuid need ei salvestu püsivalt seadme andmebaasi. Lisaks rõhutab uurimustöö pideva arendustöö vajadust, et olla manipuleerimistehnikate arengutest sammuke eespool.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 58 leheküljel, 10 peatükki, 33 joonist, 3 tabelit.

List of abbreviations and terms

WAL	Write-Ahead Log
SOP	Standard Operating Procedure
CEN	European Committee for Standardization
APFS	Apple File system
SSH	Secure Shell Protocol
SFTP	Secure File Transfer Protocol
NTP	Network Time Protocol
SSD	Solid-state drive
OS	Operating System
CLI	Command-line interface
UTC	Coordinated Universal Time
DFU	Device Firmware Update
UI	User Interface
GUI	Graphical User Interface
HFS+	Hierarchical File System Plus

Table of contents

1 Introduction 12
1.1 Novelty and scope 13
1.2 Research question..14
1.3 Outline of the thesis 15
2 Background information 15
2.1 Mobile Forensics 15
2.2 Background on iOS 17
2.2.1 iOS File System..18
2.2.2 Jailbreaking 20
3 Related work 20
4 Methodology 24
4.1 Experiment design..24
4.2 Experiment Configuration..24
5 SQLite Forensics...25
5.1 SQLite file structure...26
5.2 Data recovery 27
5.2.1 Journal mode 28
5.2.2 SQLite pragma 30
5.3 iOS database analysis...31
6 Manipulating data through GUI 34
6.1 Deleting data 34
6.1.1 Case 1 - call records 35

6.1.2 Case 2 - sms records 37
6.2 Time manipulations 39
7 Manipulation techniques in iOS database 42
7.1 Manipulating iOS data records 43
7.2 Manipulating database behaviour 46
7.2.1 Manipulating database tables 46
7.2.2 PRAGMA commands 47
7.3 Deleting databases - “Ghost messaging” 49
7.4 Creating database triggers 51
7.5 Detecting through bash history 52
7.5.1 Detecting Jailbreak 53
8 Summary 54
9 Conclusions 56
10 References 57

List of figures

Figure 1 - User data partition 19

Figure 2 - Attack tree for smartphone data manipulation [2] 21

Figure 3 - Evaluation framework to identify manupulated smartphone data [2] 22

Figure 4 - sms.db header 26

Figure 5 - SQLite file header 32

Figure 6 - Script to extract all SQLite files and its journal files 32

Figure 7 - Script to determine if database file is in WAL mode 33

Figure 8 - script to determine if auto_vacuum = 0 34

Figure 9 - scenario illustration 35

Figure 10 - CallHistory.storedata header analysis 35

Figure 11 - CallHistory.storedata files 36

Figure 12 - interactionC.db 36

Figure 13 - sms.db 37

Figure 14 - sms.db files 38

Figure 15 - com.apple.timed.plist 39

Figure 16 - CoreTime.log system vs server time 40

Figure 17 - CoreTime.log time manipulation 40

Figure 18 - CallHistory.storedata time manipualtion 41

Figure 19 - INSERT, UPDATE, DELETE 43

Figure 20 - Database record update 44

Figure 21 - manipulated data viewed from DB Browser 45

Figure 22 - DROP table ZCALLRECORD 46

Figure 23 - interactionC.db 47

Figure 24 - sms.db before force commit 48

Figure 25 - PRAGMA wal_checkpoint(TRUNCATE) on sms.db 48

Figure 26 - sms.db after wal_checkpoint(TRUNCATE) 49

Figure 27 - "Ghost messaging" 49

Figure 28 - no messages detected by iPhone Backup Extractor 50

Figure 29 - no messages detected by Magnet Axiom 51

Figure 30 - database trigger...51

Figure 31 - bash history...53

Figure 32 - Cydia, an unauthorised application 53

Figure 33 - checkra1n.dmg..54

List of tables

Table 1 - SQLite file header 27

Table 2 - System time compared to server time from CoreTime.log 41

Table 3 - Corrected times 42

1 Introduction

Mobile phones have become integral to our society, and their importance cannot be overemphasised. They have revolutionised the way we communicate, access information, and carry out daily tasks. Mobile phones are personal and can store significant amounts of data, including contacts, messages, photos, videos, location data, browsing history, and app usage data. As such, they have become a valuable source of information for a wide range of applications, from improving user experiences and personalising advertising to providing evidence in legal and investigative contexts. The data collected from mobile devices can become an important source of digital evidence. Mobile data includes any data of probative value generated by an application or transferred to the smartphone by the user [1]. Data retrieved can offer contextual clues about the user, such as whom the user knows and has communicated with, locations visited, and which activities were performed with the phone. The presence of such data can lead the user to apply manipulative techniques to update or delete data. Often without any ill intentions, but simply eliminating exposure of personal data. The user may also intentionally make changes to the data to hide involvement in criminal activities as an anti-forensic technique to mislead the forensic investigations [2]. These actions can be reversed under certain conditions, or there will be traces left from these actions that can lead the forensic investigations. The recovery of data records is dependent on database behaviour based on database settings, triggers and commands received from the application. Data records or parts of records are sometimes stored in multiple databases and can therefore be cross-referenced to each other. In this study, two different types of data are considered; data that the user can manipulate from the graphical user interface (GUI) of the device like date & time, delete data records, change personal information, and data that cannot be manipulated through the GUI like metadata of records or images, interactions with the device also known as life-data and the device interactions with the network that can only be accessed with a jailbroken phone with root access. This research aims to present data manipulation possibilities in iOS mobile phones and analyse the possibilities of detection through evidential traces.

1.1 Novelty and scope

This research analyses different data and database manipulation techniques and ways to detect or recover manipulated data. Manipulation of mobile data refers to data modification, fabrication and deletion. Data acquired from the mobile device can have immense effects in legal and criminal cases. It is relied upon for this data to be accurate and a true representation of events; however, under certain conditions, the validity of data should be strongly considered, especially when the device has already been jailbroken. It is inevitable that offenders also adapt to high-tech criminal activities and acquire knowledge on how to exploit mobile phones for their own benefit. This study aims to highlight that manipulations of data are not always detectable by forensic tools. While a manual inspection of raw data can provide a better indication that manipulation has occurred, recovery from a logical extraction is unlikely at the hands of a seasoned criminal. The extraction of allocated data records is quite straightforward, but acquiring deleted records can be difficult with the additional complexity of knowing "what" or "if" something has been deleted. When should the forensic examiner know to be critical enough to conduct a more advanced investigation? Forensic investigations are often under time pressure, and thus the case must bring a reasonable amount of doubt that data has been manipulated to conduct a more thorough analysis. To what extent deleted data can be recovered is dependent on the acquired device image. In many cases, only an encrypted logical image of the device is acquired; it is very difficult to detect any possible data manipulation or recover deleted data from the encrypted logical image. Full-file system acquisition gives access to much greater data and possibilities for deleted data recovery to an extent. While the physical acquisition or bit-by-bit copy of the device could potentially recover more records, it will be out of the scope of this study. To gain access to the full file system, the device must be jailbroken for root access. Jailbreaking is not only used for forensic purposes. Some users may prefer to jailbreak their mobile devices for advanced customisation [3]. The root access to the phone does not only permit read rights on files but also write rights. This may be beneficial in forensics or intelligence gathering, but most importantly having write rights on a full-file system allows data or database manipulation for anti-forensic purposes.

In mobile devices, SQLite is typically the database of choice. Extensive research has already been conducted in recovering deleted SQLite records in controlled experiment settings. This study will apply the existing knowledge to an iPhone 6s with iOS 12.1.3. Although the research is primarily done on one iOS version, the concept is expected to apply to other iOS versions, including Android devices. Databases store data as records, cache or logs. It was observed that not all databases have the same behaviour; this behaviour is dependent on SQLite pragma settings, database triggers and commands received from the application itself. This behaviour will be observed under different scenarios, and databases will be categorised based on different parameters. This analysis will help to understand how data in certain databases are more volatile and susceptible to data loss and how it may affect the recovery of deleted records in terms of database manipulations and understanding the traces of detection. Some existing research is done on this topic but fails to consider different types of databases, leading to false positive or false negative detection results. Mobile devices are considered "live systems" that continuously update and receive information; every interaction with the device is recorded, known as "pattern-of-life" data. As long as the mobile remains connected to the network, it continuously exchanges data through several communications protocols. It is essential to recognise that each acquisition will be different.

1.2 Research question

To what extent can data be manipulated from GUI?

To what extent can data be manipulated in jailbroken iPhone through a command line interface?

How to detect manipulations?

What determines the extent of data recovery?

How reliable is data extracted from jailbroken iPhone?

1.3 Outline of the thesis

This thesis is divided into the following chapters:

Chapter 2 – Background information on mobile forensics and iOS.

Chapter 3 – Literature review of existing research on the topic

Chapter 4 – Presents the applied methodology.

Chapter 5 – SQLite Forensics

Chapter 6 – Manipulating data through GUI

Chapter 7 – Manipulation techniques and manipulation detection in iOS database.

Chapter 8 – Summary

Chapter 9 – Conclusion

2 Background information

This section intends to provide ample background information on the topics of mobile forensics and iOS. It aims to offer a relevant understanding of mobile forensic investigations' principles, methodologies, and challenges, specifically focusing on iOS devices.

2.1 Mobile Forensics

Mobile forensics refers to the field of digital forensics that deals specifically with extracting, analysing, and preserving digital evidence from mobile devices [4]. It involves collecting and examining data from various types of mobile devices, such as smartphones, tablets, and wearable devices, to investigate and gather evidence related to criminal activities or other legal matters. Mobile forensics plays a crucial role in uncovering relevant information and evidence stored on mobile devices. Part of that is to ensure the reliability of the data acquired; is it a true representation of events and has not been tampered with. To ensure the admissibility of digital evidence in court, it is essential that the digital forensic investigator follows agreed steps and procedures for

device seizure, acquisition, examination, analysis and reporting. Maintaining a proper chain of custody is crucial for establishing the integrity and authenticity of digital evidence. This involves documenting the evidence's handling, transfer, and storage from the time of acquisition to its presentation in court. There are several standard operating procedures (SOP) and guidelines for this procedure. European Committee for Standardization (CEN) has developed a “complete end-to-end forensic investigation chain for mobile devices”, which was approved by its members on 22.02.2022 - the CWA 17865:2022 standard [5]. Estonia is one of the countries that have accepted it. Since crime has no borders, this new standard aims to unify the investigative process across law enforcement in different countries [5]. While the CWA standard gives some guidelines, it does not mandate a specific method or a tool and will give a wide action plan dependent on the examining officer’s experience. It could be argued that the standard is a little too vague and unspecific in some aspects related to mobile device seizure and evidence handling.

The CWA standard highlights that the quality of the forensic image of mobile devices holds exceptional importance for the overall evidential value of the forensic examination [5]. Logical acquisitions or encrypted backup of mobile device data may yield limited success in recovering application and user data. Full file system logical acquisitions provide a broader spectrum of collected data, log files and the possibility to recover deleted data from transaction files like wal files. The physical acquisition offers a bit-by-bit copy or a snapshot of the data contained within a mobile device, granting examiners the ability to extract additional artefacts from the device's memory. These artefacts may include deleted data or system log files that might not have been automatically decoded by forensic tools. The acquisition speed and the extraction process's duration are crucial factors that need to be carefully weighed against the type and volume of data anticipated to be recovered.

While it is recognized that it is not possible to totally preserve all evidence of mobile phones, its forensic value must not be changed during the process [5]. Since the mobile device is what is known as a "live system", it is continuously exchanging and gaining data; its numerous sensors are recording spacial awareness data, and all interactions with the device are recorded. Many digital forensics standard operating procedures (SOP) and guidelines recognise the added complexities of mobile investigations but

demand the same requirements as in digital forensics. Engaging with live systems will unavoidably lead to data alterations, heightening the likelihood of unintentional modifications by the forensic investigator. Moreover, this principle applies not only to on-site investigations but also extends to laboratory environments during post-acquisition analysis [6]. The device must be jailbroken first to acquire a full file system image; this action will make permanent changes to the device. Gruber et al. (2023) argue that when viable alternatives are lacking, forensic examiners may opt for an invasive approach to ensure more complete evidence collection and refrain from calling jailbreaking contamination as the modifications to the device are on purpose and modified objects are considered irrelevant or less relevant than the ones gained [6]. This statement is true to an extent. However, it is for a reason that mobile devices seized from a crime scene are advised not to be turned off and it becomes more so when potential anti-forensic techniques are harnessed. Jailbreaking will reboot the device and can cause data loss, cause database transaction file commits, and hide or obfuscate traces of anti-forensic techniques, etc. Jailbreaking does permit access to a greater volume of data; while jailbreaking itself may not be considered contamination. The process of jailbreaking will inevitably lead to contamination in the subtractive form [6]. Therefore, the acquisition should always be conducted from least invasive to more invasive. Further, Gruber et al. (2023) state that the forensic community should strive to develop methods for identifying digital contamination [6]. Understanding how and what kind of data is stored on the device and what actions can evoke change in data is a big part of identifying digital contamination. The field of mobile forensics is further complicated by the continuous development of new operating systems (OS), updates and new devices altogether [7]. This requires immense continuous research and proficiency to conduct mobile forensic investigations.

2.2 Background on iOS

In order to effectively perform a forensic examination on an iOS device, it is imperative to thoroughly comprehend its internal components and underlying mechanisms. This understanding is pivotal in grasping the intricacies and critical aspects involved in the forensic process. It enables investigators to ascertain the types of data that can be acquired, identify the precise storage locations of relevant data, and determine the

appropriate methods and techniques to employ in accessing and extracting the desired information [8].

iOS is a mobile operating system that is based on the Mac OS X operating system. It is considered a variant of the BSD UNIX kernel and utilizes the Mach kernel XNU, which is built upon the Darwin OS [9]. The architecture of iOS is structured in layers, with the lower layers consisting of the Core Services Layer and Core OS layer. These layers handle essential services and functions. On the upper layers, the Media and Cocoa Touch layers provide the user interface (UI) and core graphics capabilities [8]. This layered architecture allows for the separation of different functionalities within the iOS system. This layered approach also enables developers to interact with specific layers and utilize the appropriate APIs and frameworks based on their application requirements. Ultimately, the layered architecture of iOS promotes a well-structured and scalable system, contributing to the overall robustness and functionality of the operating system [8].

2.2.1 iOS File System

In iOS, the default file system used for persistent storage of data files is the Apple File System (APFS) [10]. APFS was introduced in 2016 with iOS 10.3 as a replacement for the previous Hierarchical File System Plus (HFS+). It is specifically designed and optimized for solid-state drive (SSD) storage, offering improved file system fundamentals and performance [10]. One notable feature of APFS is its built-in encryption, which operates at the file system level, providing strong data security for iOS devices. This encryption ensures that data stored on the device remains protected and inaccessible without the proper authorisation [9].

The APFS filesystem is, by default, configured as two logical partitions: the system partition and the user data partition [8]. The system partition contains the operating system and all the native applications already preloaded on the iPhone. The system partition is mounted as read-only. Jailbreak can gain full execute and write access on all partitions of iOS. The system partition may contain information about jailbreak; otherwise, little evidentiary details can be obtained from the system partition [8]. The user data partition is stored in the */private/var* directory on the device (Figure 1). It

contains user-created data. Most evidentiary information is found in this partition, particularly within the /mobile directory.

Name	Size	Date Modified
var	--	01.04.2017, 06:41
backups	--	28.10.2006, 17:07
binpack	--	07.05.2021, 22:13
buddy	--	02.05.2023, 02:37
cache	--	28.10.2006, 17:07
checkra1n.dmg	5.2 MB	02.05.2023, 02:39
containers	--	02.05.2023, 02:40
db	--	Yesterday, 14:59
dropbear_rsa_host_key	805 B	02.05.2023, 02:39
ea	--	28.10.2015, 00:18
empty	--	08.01.2019, 09:43
folders	--	08.01.2019, 09:43
installd	--	27.04.2023, 16:26
iomfb_bics_daemon	--	27.04.2023, 16:26
keybags	--	02.05.2023, 05:02
Keychains	--	03.05.2023, 05:05
lib	--	02.05.2023, 02:40
local	--	28.10.2006, 17:07
lock	--	28.10.2006, 17:07
log	--	Yesterday, 02:45
logs	--	27.04.2023, 16:28
Managed Preferences	--	27.04.2023, 16:26
mobile	--	02.05.2023, 02:40
MobileAsset	--	02.05.2023, 02:29
MobileDevice	--	27.04.2023, 16:26
MobileSoftwareUpdate	--	03.05.2023, 02:47
msgs	--	08.01.2019, 09:43
networkd	--	27.04.2023, 16:27
preferences	--	Today, 00:43
root	--	03.05.2023, 02:30
run	--	02.05.2023, 02:40
spool	--	21.10.2015, 17:01
staged_system_apps	--	02.02.2019, 20:30
tmp	--	Today, 01:04
vm	--	02.05.2023, 02:40

Figure 1 - User data partition

2.2.2 Jailbreaking

Apple designed the iOS platform with security as its core. Apple has full control over its hardware, software and services that are designed to work together for maximum security [11]. During the boot-up process a secure boot chain system, also known as a chain of trust, is employed to ensure that only trusted code is loaded onto the device. When a device is turned on, the device processor executes the Boot ROM code. This code is written on a hardware level and cannot be changed. Within the Boot ROM code resides the public key of the Apple root certificate authority (CA). This public key is used to decrypt and verify the integrity of the next stage in the boot process. At each stage of the boot process, the integrity and authenticity of the next stage are checked, ensuring that it is properly signed by Apple. The objective of a jailbreak is to break this chain somewhere to escalate privileges to gain and maintain root access on the device [12].

This research will concentrate on the semi-tethered Chekra1n jailbreak as it will be used later. Chekra1n targets an unpatchable flaw in the Boot ROM [11]. It is unpatchable as it is within the hardware itself. The vulnerable phones will remain vulnerable to this exploit no matter the iOS version. As it is semi-tethered, the phone will be able to boot up after a restart and can be used normally but will require to be jailbroken again with computer to continue using root privileges or third-party applications. The vulnerability is exploited in the DFU (Device Firmware Update) mode also known as recovery mode [11].

3 Related work

The following section describes some existing research in database manipulation techniques and how to detect traces of possible manipulations.

Pieterse, Olivier and Heerden introduce an evaluation framework to detect data manipulations in smartphones [2]. They propose a generic process or a set of conditions for data manipulation to modify, fabricate or delete data on a smartphone. Four steps are proposed; the smartphone has to be jailbroken for root access, the application's SQLite database must reside on the smartphone, the method of the manipulation should be decided: direct or off-device and lastly requires a manual reboot. Altering smartphone data in this manner can be viewed as an attack on integrity, availability, and authenticity and they illustrate it using an attack tree (Figure 2) [2].

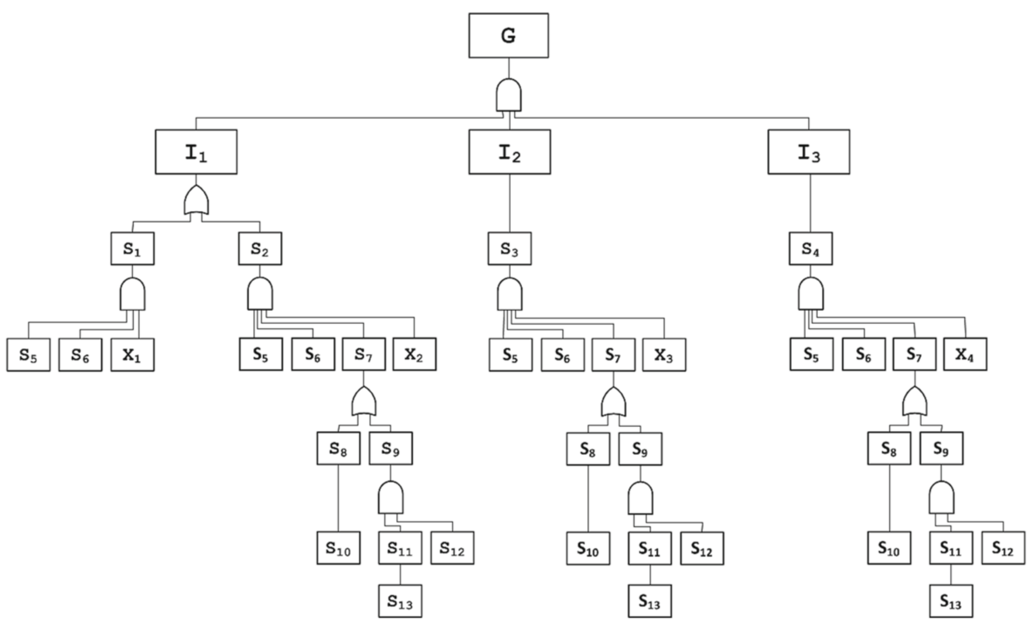

Figure 2 - Attack tree for smartphone data manipulation [2]

An attack tree is a structured and methodical way to describe various attacks against a system. The goal of the attack tree is the "manipulation of data". Intermediate goals are deletion (deletion of all data and deletion of specific data), modification and fabrication and sub-goals describe the required steps to accomplish each intermediate goal [2]. All of these actions leave various traces on smartphones that can aid in detecting and identifying any tampering or alteration of smartphone data. By examining and analyzing these traces collectively, it is possible to create a clear picture of any modifications made to the data and determine the extent and nature of the manipulation. As such, these traces can play a critical role in forensic investigations aimed at uncovering any suspicious or illegal activity related to smartphone data. For detecting

data manipulations in databases, Pieterse et al. introduce the Evaluation Framework. They describe 10 distinct indicators that each produce either positive [true] or negative [false] results (Figure 3). In an equation with all positive indicators divided by all indicators give a manipulation score that will be plotted on a probability scale from 0 to 1. While this is one way to analyse potential manipulations, it has some limitations.

No	Indicator	Measurements	Result
1	WAL File	➢ WAL file does not contain the latest stored records ➢ WAL file size < main database file size (if WAL frames < 1000)	**[true/false]**
2	Root Application	➢ ***Android***: SuperSu or Superuser application installed ➢ ***iOS***: Cydia application installed	**[true/false]**
3	OTA Updates	➢ Unavailable due to unauthorised changes (rooted/jailbroken)	**[true/false]**
4	Reboot	➢ Reboot entry logged present on the smartphone ➢ ***Android***: reboots stored in */data/system/dropbox/* folder ➢ ***iOS***: reboots stored in the */var/logs/lockdownd.log* file ➢ A reboot entry immediately follows the WAL file timestamp	**[true/false]**
5	Application Usage	➢ WAL access timestamp proceed application last used timestamp ➢ ***Android***: application logs in the */data/system/usagestats/* folder ➢ ***iOS***: application logs in the */var/mobile/Library/Preferences/* folder	**[true/false]**
6	SQLite3 Usage	➢ *sqlite3* program used ➢ *sqlite3* program access timestamp proceed WAL timestamp	**[true/false]**
7	DB Ownership	➢ Changes to the SQLite database ownership ➢ ***Android***: *UID* changed to *root* for both individual and group owners ➢ **iOS**: *UID* changed to *root* for individual owner	**[true/false]**
8	DB Permissions	➢ Changes to the SQLite database permissions ➢ ***Android***: permissions change from *-rw-rw----* to *-rw-rw-rw* for all files ➢ iOS: permissions change from *-rw-rw----* to *-rw-rw-rw* for *.db* and *.db-wal* files	**[true/false]**
9	Main Database File	➢ WAL file size < main database file size (if WAL frames < 1000)	**[true/false]**
10	Additional Settings	➢ ***Android***: USB debugging enabled	**[true/false]**

Figure 3 - Evaluation framework to identify manupulated smartphone data [2]

They do not consider that databases can have different behaviour to actions like a reboot and database deletion/recovery. Their research analyses only one iOS database, the sms.db database. The sms.db will be analysed in later parts of this thesis as it was found to have unique characteristics specifically to these actions. The suggested indicators do match the sms.db database but not most other databases. Most databases in iOS 12.1.3 do not require a reboot to recreate a database, this would lead false negative results based on this evaluation framework. This discrepancy can also be due to different iOS versions having updated database settings. Further, it can be criticised that more positive indicators do not necessarily mean a higher probability of data manipulation. The likeliness should be individually analysed based on the indicator. They also do not consider that timestamps can be manipulated in a jailbroken mobile device.

Albano et al wrote a study on constructing a false digital alibi in Android OS [13]. Despite their research on Android, this is just as applicable to iOS. Their work was aimed to highlight how it could be possible to artificially create a false digital alibi. Digital evidence can play a pivotal role in determining the conviction or acquittal of a suspect. One of the previous studies [14] emphasized the need for caution by courts when assessing digital evidence's admissibility and probative value. The study highlighted that individuals, regardless of their level of expertise, can utilize software automation to generate digital traces on a computer. These traces can be virtually identical to those produced by regular user activity, making them difficult to distinguish through post-mortem analysis [13]. The utilization of automation can be exploited to fabricate a false digital alibi, creating a collection of seemingly "reliable" digital evidence. They state that any unwanted traces produced by the tools were simply removed during the sanitization process. This is highly relevant in this study as the automation that is being referred to is a similar set of commands and techniques applied in this paper, just scripted and automated.

The consideration of database behaviour and SQLite pragma settings as a decisive element in the recovery of deleted data has been relatively limited in the existing body of research papers. Christian Meng and Harald Baier go in-depth into SQLite pragma settings that affect the erasure behaviour of data records and how that relates to data recovery [15]. They introduce a tool, “bring2lite”, and discuss its superiority compared to other recovery tools available. However, the tool is also limited to recovering within the database files; once the data record is rewritten by new data or the database is vacuumed or truncated, recovery is no longer possible. bring2lite primarily focuses on data recovery within the Write-Ahead Log (WAL) file and unallocated space within the SQLite database file. They define the relevant pragmas as secure_delete, auto_vacuum and journal_mode [15]. They describe in detail how these parameters affect database behaviour and thus data recovery. While they recognise auto_vacuum as an important pragma parameter in their experiments, they have set auto_vacuum to 0 (none) in all of the experiments. It would be interesting to see how auto_vacuum "incremental" or "full" would test out in bring2lite tool. They also do not consider scenarios in which databases could be checkpointed by events like a reboot or application commands.

4 Methodology

The aim of this study is to analyse the intended functions of iOS databases, specifically how data is recorded, edited, and deleted while considering SQLite pragma settings, triggers and other factors affecting database behaviour. This serves as a pre-study for the main focus, which is examining how these databases can be manipulated and detected. This research adopts an experimental research methodology to achieve its objectives.

4.1 Experiment design

Experimental research involves the deliberate manipulation of variables in a controlled setting to study cause-and-effect relationships. In this study, will be conducted experiments to simulate various scenarios related to data manipulation in iOS databases, by controlling the variables, like SQLite pragma settings, and systematically manipulating them to observe the effects on data storage, editing, and deletion. This experimental approach aims to gain a deeper understanding of the potential manipulation techniques within iOS databases, including how manipulations could be detected. Additionally, it seeks to highlight the potential problems that may arise if these manipulations go undetected.

4.2 Experiment Configuration

The experiments are conducted on an iPhone 6s running on iOS 12.1.3. To ensure a clean slate, the device is initially subjected to a factory reset, resetting all settings to their default values. In order to gain access to the file system and perform extractions, the device is subsequently jailbroken using the Checkra1n jailbreak tool. Once jailbroken, the Cydia application is installed, which provides an OpenSSH server package. This package is utilized for the manual extraction of the database files. OpenSSH also enables a Secure File Transfer Protocol (SFTP) connection, allowing for a visual representation of the file system. This connection is utilized for live analysis of data transactions by observing date/time modifications to the databases and changes in file sizes. It is important to note that all records on the device are artificially created for the purpose of this study. File manipulations are carried out using a command line

interface Terminal. iOS does not come with an extensive set of pre-installed tools. However, unlike Android, iOS includes the sqlite3 program as a built-in component [3].

The scope of this research is limited to the private/var/mobile directory, which houses a significant amount of user data with forensic relevance. All databases within this directory are extracted using a bash script for subsequent analysis. While the most common file extensions for SQLite databases are .sqlite, .sqlite3, .db, .db3, .s3db, and .sl3 [16], it is worth noting that SQLite allows for flexibility in file extensions or even no extension at all. Therefore, file filtering is conducted based on the initial bytes of the file represented in hexadecimal format. The extracted databases from iOS 12.1.3 are studied based on the pragma settings, triggers and unique behaviour recognised by observing the databases in different situations. While the behaviours stemming from applications in databases is outside of the scope in this study, it presents how important it is to study each iOS version as these often have critical importance in how data could be recovered.

After the preliminary study on databases, various experiments are conducted to explore data manipulation through the graphical user interface (GUI) in an iPhone and the Terminal command line interface. Chapter 7 delves into the exploration of creating a false digital alibi inspired by Albano et al's research [13] and by modifying data records and manipulating databases to the extent that they no longer store data in their conventional form, while still allowing the application to function as intended.

5 SQLite Forensics

SQLite forensics involves analysing SQLite databases to extract and interpret relevant information for investigations. SQLite database is a popular choice in a wide range of applications, including iOS and Android mobile devices, making it a valuable source of information in mobile forensics [16]. From a forensic perspective, it is crucial to grasp the underlying mechanisms of data storage within mobile device databases and comprehend the implications when data is deleted or modified. The process of extracting forensic data from iOS devices through logical extraction relies significantly

on the organisation of data within the file system, particularly in the form of SQLite database files [7]. The following chapter will provide fundamentals of SQLite database functions and relevant aspects of database forensics and provide an analysis of iOS databases.

The complete state of the SQLite database is contained in a single file on a disk called “the main database file”. In SQLite database, the data in transaction is written to a temporary journal or WAL file. This is to ensure data integrity and the possibility of rollback in case of potential write errors.

5.1 SQLite file structure

At the logical level, data is organized and stored in tables, following a row and column format. However, from a physical perspective, the data is actually stored in pages organised in B-tree structure.

When a database is created, certain configurations that cannot be changed afterwards must be adjusted; they are written to the header of the database [15]. The header consists of the first 100 bytes of the file in hexadecimal representation and contains metadata about the database, such as format, page size and certain pragma settings parameters [16] (Figure 4). The first page of the database contains the database header and the master table.

```
janikahirvi@Janikas-iMac Documents % hexdump -C sms.db
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  10 00 02 02 00 40 20 20  00 00 00 0b 00 00 00 41  |.....@  .......A|
00000020  00 00 00 00 00 00 00 00  00 00 00 4b 00 00 00 04  |...........K....|
00000030  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 0b  |................|
00000060  00 2e 24 80 05 00 00 00  03 0f f1 00 00 00 00 41  |..$.......?....A|
00000070  0f fb 0f f6 0f f1 00 00  00 00 00 00 00 00 00 00  |.?.?.?..........|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

Figure 4 - sms.db header

The database header provides valuable information about the parameters and settings of the database. The first 16 bytes of the database file hold the SQLite file signature. Every valid SQLite database begins with in hexadecimal: 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00 as presented in Figure 4. Table 1, shows additional information that is stored in the SQLite database header.

Offset	Description	sms.db
0	The header string	SQLite format 3
16	The database page size in bytes	4096
18	File format write version (02 - wal, 01 - legacy journal)	wal
19	File format read version (02 - wal, 01 - legacy journal)	wal
28	Size of the database file in pages	65
52	The page number of the largest root b-tree page when in auto or incremental vacuum, 0 otherwise	0 - false
64	True (non-0) for incremental-vacuum mode. False (0) otherwise	0 - false
96	SQLITE_Version_number	3024000

Table 1 - SQLite file header

The header is followed by the SQLite master table, which holds essential information about the tables and index schemas of the database [15]. SQLite stores data in B-tree pages, which are similar to other database systems. The B-tree structure allows for efficient searching and sorting of data. In a database, the data is divided into fixed-size pages, which are typically 4 kilobytes or more accurately 4096 bytes in size [17]. Each page can contain multiple rows of data, along with their corresponding columns. When data is inserted into a table, it is distributed across multiple pages based on the underlying storage structure. The pages are allocated dynamically to accommodate the data, and as the amount of data grows, additional pages may be allocated to ensure sufficient storage capacity [17].

5.2 Data recovery

For time and resource reasons typically, the SQLite does not delete database records instantly, but they are marked for deletion until an event that will commit the changes and data is “deleted”. Based on the database settings it is determined what happens to the deleted data next. Some databases will store deleted data for a certain period of time. For example, the iOS Photos app will store deleted data for another 30 days in a deleted folder until it becomes unavailable for the user or is recovered during this time. While in other application databases the data becomes immediately unavailable to the user and recovery is only possible through previous backup or specialised tools. In

many cases, this means that the index for that data is lost, but data hangs around there until it gets written over by new data. In the field of recovering deleted SQLite records, there are both commercial and open-source tools available. Despite specialised recovery tools, the recovery is still dependent on the SQLite database settings, how much time has passed and if the phone has been rebooted in the meantime. The quality of the forensic image of mobile devices holds exceptional importance for the overall evidential value of the forensic examination [5]. There are several parameters that affect how data is stored and erased from the database.

5.2.1 Journal mode

The journal file is used to maintain database integrity and to record changes made to the database so that they can be rolled back in the event of failure. The journal_mode pragma determines how the journal file is used. There are different journal_mode options: DELETE, TRUNCATE, PERSIST, MEMORY, WAL and OFF [17].

When the SQLite database is set to a WAL journal mode, two files .db-wal and .db-shm files are also created additionally to the main database file. Write-Ahead Log (WAL) file is a form of a cache or a roll-forward journal that records data that has been committed but not yet written to the main database [18]. In WAL mode the database engine does not touch the database file when new data gets recorded, edited or deleted but instead gets stored in a separate WAL file. Data remains in the WAL file until it gets committed by the checkpoint event. The checkpoint occurs automatically once the WAL file reaches a certain size, by default, it is 1000 pages. Until the checkpoint occurs, the main database file does not contain the most recent information.

Forensically it would be very important to analyse the main database files separately from WAL files as the main database file could contain information that has been assigned to be deleted or edited but not yet committed to the main database file [19]. Which after the commit would be lost. The WAL files can get very large and can contain data from a long period and thus cannot be ignored in the investigations. The WAL files are difficult to read without a specialised tool that can parse the WAL files, but some forensic tools automatically process the WAL risking with loss of data. One such way for this to happen is when a main database file is opened and closed in an

extracted full file system copy the device, the .db-wal and .db-shm files disappear as they get automatically merged into the main database file [19]. This is a permanent change within this extraction; data gets written over to the main database. As a side note this does not happen inside the devices' file system but only when the files are extracted from the device. For the merge to happen these 3 files (.db, .db-wal and .db-shm) must be inside the same folder. Many forensic tools incorporate the associated WAL file as a part of the analysis process, to identify as many unmerged and unique records as possible and do not allow the merge process to happen.

One challenge in investigations is to identify that there are missing records within the database. The SQLite database table uniquely identifies each row with a key, known as primary key (PK). The numbering starts with a value "1" and autoincrements with every record that is created. If the PK numbers don't run contiguously, it is a good indication that some records have been deleted [19]. Since data is recorded consecutively generally the timestamps of deleted records fall between the existing records. It can give some idea of when data was recorded. This is true in most cases even when date and time have been manipulated by the user, but there are anomalies to this as well like when calls are made to a phone that is in flight mode or the phone is in a low signal area [19].

The remaining journal modes in SQLite are regarded as legacy journals and primarily serve as roll-back journals [20]. In these modes, when a data write occurs in the main database file, the old record is retained in the corresponding -journal file until the transaction is finalized. The specific types of journal modes, namely DELETE, TRUNCATE, PERSIST, and MEMORY, determine the behavior of the -journal file after the data records have been committed to the main database file. During the analysis of iOS, it was observed that all -journal files were configured in the DELETE mode. In the DELETE mode, the rollback journal is deleted at the end of each transaction. This means that when a delete operation is performed, it automatically triggers the transaction to commit, resulting in the deletion of the rollback journal [20].

5.2.2 SQLite pragma

The database behaviour is highly dependent on SQLite database settings known as PRAGMA settings. These commands often determine how data is stored on the database, its transactional files, and how data is erased. This paper will concentrate only on the PRAGMA settings that have potential forensic implications. Before a database file is created, among other settings pragmas like secure delete, auto vacuum and journal mode must be selected.

secure_delete

SQLite provides a delete command to the user, in which a user can choose which records will be deleted. The secure_delete pragma is used to control how content is deleted from the database. By default, when content is deleted, it is typically marked as unused rather than immediately erased from the database files. When the secure_delete setting is enabled, deleted content in SQLite databases is overwritten with a sequence of 0 byte values [20]. The secure_delete pragma can be configured off (0), on (1) and FAST (2). By default, the secure_delete option is turned off to enhance performance by minimizing CPU cycles and disk writes [20]. Applications that wish to avoid leaving forensic traces after content is deleted or edited should enable the secure_delete pragma or run the VACUUM command after the delete or update. The FAST setting is an intermittent setting between “on” and “off”. When secure_delete is set to “FAST” SQLite will overwrite deleted content with zeros only if doing so does not increase the amount of disk writes (I/O) [20].

auto_vacuum

As mentioned, when content is deleted from an SQLite database, the content is usually not erased from the database, but the space is marked as being available for reuse. When large amounts of data are deleted, it leaves behind empty space, or “free” database pages, making the database file much larger than strictly necessary [20]. Running the command "VACUUM" will rebuild the database file so that each table and index is stored contiguously, effectively writing over previous content and preventing traces of deleted content from being recovered by forensic analysts, and thus reducing the size [20]. This does not however change the primary key value of the

data record but simply reorganises data within the database. There are 3 different types of vacuuming parameters: "none" (0), Full (1) and Incremental (2). The default setting for auto-vacuum is "none" or disabled. When auto-vacuum is disabled, and data is deleted from the database, the file size remains unchanged. However, the space previously occupied by the deleted data becomes available for storing new data. If the database is in full auto-vacuum mode, the deleted data is moved to the bottom of the page and removed at commit. It does not however defragment the database, in fact, it can be made worse since the file is moved around in a database. The VACUUM command would defragment the database. The incremental mode the auto-vacuum is enabled but must be activated by the application.

5.3 iOS database analysis

This chapter will analyse in depth what possibilities there are to recovering deleted data and which database settings can affect the possibilities of recovery. All databases on the user partition private/var/mobile directory will be pulled and categorised based on the SQLite database settings and analysed based on data volatility. This is relevant as in some databases a reboot can cause a commit on a database and cause loss of data. Forensic investigations are typically time-limited and while in some cases it might be possible to recover deleted records - if there is no knowledge of them there is no knowledge to look for them. Not every database might leave traces of data erasure. Further it will also be analysed to which extent can the database settings be manipulated to either as an anti-forensic technique or as a forensic technique to make databases more stable thought out the investigation (latter is still a hypothesis, I fear its not too useful as its possible only after phone is jailbroken and therefore no longer reasonable).

This section is a preliminary investigation of iOS 12.1.3 databases to analyse how different databases function and store data. This is relevant as data storage is a complex process where parts of data or traces of actions are written to multiple databases. Understanding how specific databases are intended to function will help understand how anti-forensic techniques can be implemented and, more importantly, how to detect them.

This analysis considers all databases from the user partition or /var directory since most of the user-created data records are located there. From the /var directory, all databases are extracted through bash script to a local computer for analysis. Every SQLite database starts with the same 16 bytes in hexadecimal, which also translates to "SQLite format 3." in binary; this is the file signature of SQLite databases. This can be seen in Figure 5 with a red line

```
janikahirvi@Janikas-iMac Databases % hexdump -C CallHistory.storedata
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  10 00 02 02 00 40 20 20  00 00 00 05 00 00 00 12  |.....@  ........|
00000020  00 00 00 00 00 00 00 00  00 00 00 0e 00 00 00 04  |................|
00000030  00 00 00 00 00 00 00 11  00 00 00 01 00 00 00 00  |................|
00000040  00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 05  |................|
00000060  00 2e 24 80 0d 0c cc 00  0f 04 d2 00 0f 0b 0c d4  |..$...?...?....?|
00000070  0b 76 0c 75 0a d3 09 c8  09 47 08 95 07 cb 07 33  |.v.u.?.?.G...?.3|
00000080  06 bf 06 2c 05 9f 05 22  04 d2 00 00 00 00 00 00  |.?.,...".?......|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Figure 5 - SQLite file header

A script was written to extract all SQLite databases and it's accompanying journal files (Figure 6). The script looks for the file signature, and if it matches, it will also look for its accompanying journal files and extracts them to a new folder. All together, 457 files were extracted, out of which 189 were databases and the rest were journal files. The information can be gained from the file header to analyse which journal modes have

```
6     # Loop through all files in directory
7     find "$src_dir" -type f | while read file; do
8
9       # Check if the file is an SQLite database file
10        sqlite_signature=$(xxd -l 16 -p "$file")
11        if [[ $sqlite_signature == "53514c69746520666f726d6174203300" ]]; then
12        cp "$file" "$dest_dir"
13
14        # Check for any journal or temporary files associated with the database
15        journal_file="${file}-journal" wal_file="${file}-wal" shm_file="${file}-shm"
16
17        # Copy any journal or temporary files to the destination directory
18        for file in "$journal_file" "$wal_file" "$shm_file"; do
19            if [ -f "$file" ]; then
20              cp "$file" "$dest_dir"
21            fi
22        done
23    fi
24  done
```

Figure 6 - Script to extract all SQLite files and its journal files

been used. If the value at offset 18 and 19 in hexadecimal representation is "02", the file is in WAL journal mode; if it is "01", it is in legacy journal mode. The value presents the file format read and write version [17]. To sort the databases, a script was made to look for the value at these offsets (Figure 7). The blue line in Figure 5 presents that both bytes are "02" and the "CallHistory.storedata" database in WAL mode. To emphasise the importance of journal mode, it determines how most recent data is stored and greatly affects the recovery of the records. It was determined that out of 189 databases, 151 were in WAL mode, and 38 were in legacy journal mode.

```
9   # Loop through all files in directory
10  find "$src_dir" -type f | while read file; do
11  # Check if the bytes at offset
12    bytes=$(xxd -p -s 18 -l 2 "$file")
13    if [[ $bytes == "0202" ]]; then
14      cp "$file" "$dest_dir"
15    fi
16  done
```

Figure 7 - Script to determine if database file is in WAL mode

Most of the legacy -journal databases were not updated often or databases that provided some functions, for example, the emoji.db contains all emojis that can be used in chat functionality.

There are no databases with iOS secure delete turned on. Several 3rd party messaging applications like Telegram, WhatsApp, Snapchat, Discord and ProtonMail were installed, but neither of them had secure delete pragma turned on. Then, it was observed what auto-vacuum modes were used. First, a script was made to determine if "PRAGMA auto_vacuum;" equalled 0 (Figure 8). It was found that 39 out of 151 databases were in auto_vacuum = 0 mode. This script was repeated for other modes. There were 21 databases with auto_vacuum = 1 mode and 91 were in auto_vacuum = 2 mode. This information is relevant when data records are recovered from the WAL file.

```
 7  # Loop through all files in directory
 8  for file in "$src_dir"/*.db; do
 9      # Check if the file is with auto-vacuum 0
10      if sqlite3 "$file" "PRAGMA auto_vacuum;" | grep -q "0"; then
11          filename=$(basename "$file")
12          cp "$file" "$dest_dir/$filename"
13          echo "Copied: $filename"
14      fi
15  done
```

Figure 8 - script to determine if auto_vacuum = 0

6 Manipulating data through GUI

This chapter will analyse how data can be manipulated through the graphical user interface (GUI) without jailbreak in iOS. The actions are generally limited to the functionalities provided by the application. Detection of missing records is generally relatively easy to tell, and recovery depends on database parameters. The analysis of data traces from WAL in this section is conducted manually in hexadecimal representation without any specialised forensic tools. The database is viewed with DB Browser for SQLite.

6.1 Deleting data

The recovery of deleted data is dependent on database settings and possible triggers. In this section is observed two separate cases where the record is created, deleted, and then the device is rebooted (Figure 9). A scenario is created where the user aims to make the data unrecoverable. At the end of each case, the traces of the user's actions left in the device are analysed. The device is factory reset for this experiment to ensure no previous database manipulations are applied. The database header is analysed using a script with the most relevant parameters. These databases are chosen to illustrate how database settings and behaviour matter in a forensic investigation. It is not always possible to recover data from the WAL file or recovery is more complicated.

Figure 9 - scenario illustration

6.1.1 Case 1 - call records

Scenario: The user makes a Facetime call to phone number +372 501 6463, deletes the call through the GUI application and reboots the device. A potential connection to this phone number is being investigated. The call records are primarily being recorded in CallHistory.storedata database in var/mobile/Library/CallHistoryDB. Below is attached an SQLite database header analysis (Figure 10).

```
janikahirvi@Janikas-iMac Documents % bash analyse.sh
SQLite Database Header Analysis
------------------------------
CallHistory.storedata
Date Created: 2023-05-11 17:20:25.000000000 +0300
Last Modified: 2023-05-11 17:20:25.000000000 +0300
Last Accessed: 2023-05-12 20:16:01.837808000 +0300
SQLite Version: 3.37.0
Journal mode: wal
Page Size: 4096 bytes
Total Pages: 18
Freelist Pages: 0
Secure Delete: false
Auto_vacuum: incremental

Triggers:
No triggers found in the database.
```

Figure 10 - CallHistory.storedata header analysis

Expectations: It was observed that a reboot of the device causes a Checkpoint event on the CallHistory.storedata database and the -wal file gets committed to the main database. This is an unusual behaviour; presumably, the database gets a force wal_checkpoint() or wal_checkpoint(TRUNCATE) command from its application when the database connection is closed and thus rebooted. There are no SQLite pragma settings to perform this operation automatically. As the database is in incremental vacuum mode, the file size will be reduced to reclaim any unused space at a checkpoint (Figure 10). Since all of the indexed data records are committed to the database after

the checkpoint event, the file size is reduced to zero. Therefore, if a data record were made and deleted before it was committed to the main database, it could be unrecoverable from the wal file.

Analysis: At the time of the investigation, it can be observed that the CallHistory.storedata-wal file size is 0 bytes (Figure 11); therefore the -wal file has been recently committed and reduced in size. Since the wal file is empty, recovery can only be attempted from the main database file CallHistory.storedata.

Figure 11 - CallHistory.storedata files

When the database is opened with DB Browser for SQLite, there is no record of the phone number in the ZCALLRECORD table where all call records are recorded. Since in this table is recorded metadata related to the call, this information is effectively lost. The phone number gets also recorded in the same database in another table called ZHANDLE, which records all phone numbers contacted or received together with the unique Z_PK primary key. From this table can be determined how many times the phone number has been contacted and can roughly set the timeframe based on the existing call records. It is determined that this number has been called only one time.

Under the var/mobile/Library/CoreDuet folder there is another powerful database called interactionC.db that could help recover recent calls. In the table ZCONTACTS is recorded all contact numbers or accounts even if all interactions with a specific contact are deleted. This table records when the first and last messages were sent and when the first and last calls were received or answered, including the total interaction

	Z_PK	ZOUTGOINGRECIPIENTCOUNT	ZDATE	FIRST_OUTGOING	LAST_OUTGOING	ZIDENTIFIER
6	6	1	2023-05-11 17:13:53	2023-05-11 17:13:53	2023-05-11 17:13:53	+3725016463
7	7	1	2023-05-11 17:20:01	2023-05-11 17:20:01	2023-05-11 17:20:01	+3726973845
8	8	0	2023-05-12 12:46:53	NULL	NULL	tele2

Figure 12 - interactionC.db

count with a contact. From Figure 12 can be found that it was an outgoing call made on 11.05.2023 at 17:13 to the number +372 5016 463.

Conclusion: CallHistory.storedata has some unique characteristics presumably derived from the native Telephony application for calling. If there was another database with similar characteristics that checkpoint at reboot and data are recorded only in one database and the application does permit deletion through GUI, permanently deleting data could be possible through GUI.

6.1.2 Case 2 - sms records

Scenario: The user sends a message to phone number +372 501 6463, deletes the message through the GUI application and reboots the device. A potential connection to this phone number is being investigated. The messages records are primarily recorded in sms.db database in var/mobile/Library/SMS folder. Below is attached an SQLite database header analysis received with a script (Figure 13).

```
janikahirvi@Janikas-iMac Documents % bash analyse.sh
SQLite Database Header Analysis
------------------------------
sms.db
Date Created: 2023-05-10 03:22:55.000000000 +0300
Last Modified: 2023-05-13 03:00:00.000000000 +0300
Last Accessed: 2023-05-13 09:52:54.837235000 +0300
SQLite Version: 3.37.0
Journal mode: wal
Page Size: 4096 bytes
Total Pages: 65
Freelist Pages: 0
Secure Delete: false
Auto_vacuum: None

Triggers:
after_delete_on_chat_message_join|chat_message_join
after_delete_on_attachment|attachment
after_insert_on_message_attachment_join|message_attachment_join
after_delete_on_chat_handle_join|chat_handle_join
after_insert_on_chat_message_join|chat_message_join
after_delete_on_message|message
update_message_date_after_update_on_message|message
add_to_sync_deleted_messages|message
after_delete_on_chat|chat
delete_associated_messages_after_delete_on_message|message
before_delete_on_attachment|attachment
add_to_sync_deleted_attachments|attachment
add_to_deleted_messages|message
after_delete_on_message_attachment_join|message_attachment_join
```

Figure 13 - sms.db

Expectations: As it can be seen from the Figure 13 sms.db is in wal journal mode with auto vacuum none. The wal file is intact, and a straightforward process is expected in recovering data records through the wal file.

Analysis: sms.db has a lot of triggers that seem to be affecting how deleted records are managed within the database. In this case, this seems to be primarily for statistical purposes. The trigger is used to remove the content and most of the metadata at delete within the database but keep the count of the records deleted in another table. A database write is made as a result of the record being deleted. This does not hinder or aid in data recovery; the chances of recovery depend on the wal file. Reboot will not affect sms.db nor its journal files.

In the table "handle" is recorded all unique phone numbers or contact accounts received or sent from, but unlike in CallHistory.storedata, the sms.db does not store contact records together with Z_PK values. Therefore, based solely on sms.db can be detected that the +372 501 6463 does exist in the table "handle", but the potential timeframe cannot be identified. Because in sms.db the auto vacuum is turned off (none) even after the checkpoint event the -wal file will never get empty (Figure 14). The data remains in the database without indexes until it gets overwritten by new data. When analysing the -wal file in hexadecimal representation the phone number +372 501 6463 can be identified multiple times. This is because of the numerous database indices that store this data. Most forensic tools that have the ability to read WAL files could recover deleted message records.

sms.db	266.2 KB	Today, 03:00
sms.db-shm	32.8 KB	11.05.2023, 17:20
sms.db-wal	1.1 MB	Today, 09:49

Figure 14 - sms.db files

Conclusion: Recovering deleted data records can be a relatively straightforward case when employing appropriate forensic tools that are capable of analysing wal files and unallocated space within the database files.

6.2 Time manipulations

Creating a correct timeline of events is essential in any forensic investigation. In iOS the system time is by default automatically set but the date and time can be adjusted from the GUI. Depending on the forensic tool used the detection of time manipulations from simple logical acquisition is not always possible. For example, Magnet Axiom does not record the original primary key number and assigns its own Item ID for each unique artifact recovered. This does not consider the true chronological order of events. The records can either be sorted by Item ID or data record date, which can create an incorrect timeline.

In iOS databases the timestamps are recorded in "Mac absolute time", also called "CFAbsoluteTime" and "Cocoa Core Data timestamp". The time is measured in seconds or nanoseconds and the Mac absolute time is measured from 01.01.2001, GMT. The more commonly used Unix timestamp or Unix epoch time is measured in seconds since 01.01.1970. While most databases and files in iOS use Mac absolute time, some files like recents.db use Unix timestamp, it is not always consistent. Timestamps are given in UTC time; date and time must be converted to local time.

To identify if the time is set automatically or manually at the time of the acquisition, can be seen from the com.apple.timed.plist file under /var/db/timed folder. There are two variables TMAutomaticTimeOnlyEnabled and TMAutomaticTimeZoneEnabled; if they are both set to "true" the time and time zone is set automatically (Figure 15).

```
com.apple.timed.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "ht
<plist version="1.0">
<dict>
        <key>TMAutomaticTimeOnlyEnabled</key>
        <true/>
        <key>TMAutomaticTimeZoneEnabled</key>
        <true/>
</dict>
```

Figure 15 - com.apple.timed.plist

iOS gets the corrected time over Apple’s Network Time Protocol (NTP) server and is repetitively synchronizing the time offset of system time relative to the server time. This can be seen from the CoreTime.log file in /var/mobile/Library/Logs/CoreDuet

folder (Figure 16). CoreTime.log file shows the set system time in readable date and time format and the continuous synchronization process of the device matching its own Real Time Clock (RTC) time and Coordinated Universal Time (UTC) [11]. The server time is set in UTC time written in Mac absolute time.

```
2023-01-01 00:25:35.30 +0200 timed[3773]:
_utc_sf_cov:4.766844373019637e-08;_rateSf:0.9999785635083793;_rtc:2
_utc_var:9.125654053532385e-05;type:31;_utc:694221937.8623116;_sf_v
454e-11
```

Figure 16 - CoreTime.log system vs server time

The system time can be compared to the server time to detect if they appoint at the same time. In Figure 16, the system time is set as 01.01.2023 00:25:35 in UTC +2, and the server time converted from Mac absolute time is 01.01.2023 01:25:37 in UTC +2. There is a 1-hour difference between them. This is an indication that system time is set 1 hour earlier than the server time. In Figure 17 can be observed a CoreTime.log file where there is a conflict in time; a past date is turned into an earlier date.

```
2023-01-01 01:11:02.48 +0200 timed[3773]: bb_want_rtc_s:inf;type:1
2023-01-01 01:11:06.19 +0200 timed[3773]: bool:1;type:17;src:26;rt
2023-01-01 01:11:06.19 +0200 timed[3773]: reason:0;type:14;src:26;
2023-01-01 01:11:06.21 +0200 timed[3773]: bool:1;type:16;src:26;rt
2023-01-01 01:11:06.22 +0200 timed[3773]: type:20;rtc_s:2216503.39
2022-12-31 08:11:10.43 +0200 timed[3773]: t_s:694159870.4016006;ad
2022-12-31 08:11:10.44 +0200 timed[3773]: type:20;rtc_s:2216503.43
2022-12-31 08:11:10.44 +0200 timed[3773]: t_s:694159870.4439665;ad
2022-12-31 08:11:10.44 +0200 timed[3773]: type:20;rtc_s:2216503.43
2022-12-31 08:11:10.44 +0200 timed[3773]: type:6;src:10;rtc_s:2216
```

Figure 17 - CoreTime.log time manipulation

These two examples may or may not be evidence of data tampering. Changing date and time is a functionality provided by the operating system. It becomes an illicit behavior if by changing the time the user tries to cover up the order of events as an anti-forensic technique. The *CoreTime.log* file keeps a record of one day, after which the file is archived under */var/mobile/Library/Logs/CrashReporter/Retired* folder for up to 7

days. The CoreTime.log can be cross-referenced with other data records in case a manipulation of time is suspected but only for up to 7 days.

In the following Figure 18 is a screenshot from *CallHistory.storedata*. ZDATE column shows the system time, these values will be compared to the server time.

	Z_PK	ZANSWERED	ZCALLTYPE	ZORIGINATED	ZREAD	ZDATE	ZDURATION	ZISO_COUNTRY_CODE	ZSERVICE_PROVIDER
43	62	0	1	0	1	2023-01-01 13:29:20	0.0	ee	com.apple.Telephony
44	63	0	1	1	1	2023-01-01 13:06:18	0.0	ee	com.apple.Telephony
45	64	0	1	0	1	2023-01-01 13:08:57	0.0	ee	com.apple.Telephony
46	65	0	1	0	0	2023-01-01 13:12:21	0.0	ee	com.apple.Telephony
47	66	0	1	0	0	2023-01-01 14:28:29	0.0	ee	com.apple.Telephony

Figure 18 - CallHistory.storedata time manipualtion

The closest timestamp in *CoreTime.log* will be chosen for the call record. This leaves some variability when exactly the times were changed.

Nr	System time UTC +2	Server time in Mac absolute time (UTC)	Server time UTC +2	Time difference
1	01.01.2023 13:27:09	694265229.758649	01.01.2023 13:27:09	0
2	01.01.2023 13:06:25	694265612.7223855	01.01.2023 13:33:32	0:27:07
3	01.01.2023 13:11:32	694265919.069291	01.01.2023 13:38:38	0:27:06
4	01.01.2023 14:26:55	694268815.5969685	01.01.2023 14:26:55	0

Table 2 - System time compared to server time from CoreTime.log

Based on the results from Table 2, we can be sure that at least the call records with Z_PK 63, 64 and 65 have been turned 27 minutes backwards and Z_PK's with 62 and 66 are likely in correct time (Table 3).

Z_PK	Date/Time
63	01.01.2023 13:33:25
64	01.01.2023 13:36:03
65	01.01.2023 13:39:27

Table 3 - Corrected times

An interesting observation was made regarding the timestamps of iMessages, which consistently reflected the correct time even when the system time was manipulated. iMessages are timestamped using Apple's NTP server time. Therefore, during forensic analysis, it is crucial to avoid converting the timestamps of iMessages as they are already recorded in the accurate time.

7 Manipulation techniques in iOS database

The preceding chapter delved into the analysis of data recovery and detection from unmanipulated databases with their intended functionalities. This chapter analyses how databases can be manipulated to behave differently than originally intended to potentially undetectable lengths. Jailbreaking is not only used for forensic purposes; some users may prefer to jailbreak their mobile devices for advanced customisation [3]. The root access to the phone does not only permit read rights on files but also write rights. This may be beneficial in forensics or intelligence gathering but most importantly having write rights on a full file system allows data or database manipulation for anti-forensic purposes. If the seized mobile device is already jailbroken, acquiring data is much easier for the forensic examiner [3]. The aim of these experiments is to present how little specialised knowledge is required to manipulate databases, and there is no requirement for specialised tools. Users who are accustomed to jailbroken phones often possess a higher level of familiarity and knowledge about the iOS operating system. This familiarity stems from their experience in modifying and customizing their devices beyond the manufacturer's intended boundaries. As a result, these individuals may have a deeper understanding of iOS internals, system

vulnerabilities, and alternative methods of app installation and system manipulation. In the context of criminal activities, it is evident that offenders continuously evolve their techniques to engage in high-tech criminal acts [3]. Jailbreaking, which was initially considered a niche activity, has attracted the attention of individuals involved in various forms of illegal activities. These offenders leverage their familiarity with jailbroken devices and their knowledge of iOS to engage in sophisticated, technology-driven criminal endeavours [3].

Different scenarios will be carried out to illustrate different manipulation techniques in jailbroken phones and what traces could be observed from these actions. All experiments are conducted through SSH connection to the device through the command line interface. iOS does not come with many tools installed on its system. Unlike Android, iOS is equipped with the sqlite3 program pre-installed [3]. This program enables direct access to the SQLite database, allowing for the modification of existing data, the creation of new data, and the selective removal of specific or all data.

Detection of any manipulation, particularly when performed by a root user, should lead to the exclusion of the mobile data records from the list of admissible evidence. As demonstrated by these examples, detection is not always feasible or straightforward or can be reversed by a reboot that will recreate the database file. In this case, investigators find a database with fewer records than expected when compared to the other records on the device.

7.1 Manipulating iOS data records

In jailbroken phones, users have elevated privileges and can access the underlying system files and settings, including the ability to manipulate data through the command-line interface (CLI). This opens up additional possibilities for directly interacting with the SQLite database and performing data manipulation operations. By

```
sqlite3 database.db "INSERT INTO table (column1) VALUES ('value1');"

sqlite3 database.db "UPDATE table SET column1 = 'new_value' WHERE condition;"

sqlite3 database.db "DELETE FROM table WHERE condition;"
```

Figure 19 - INSERT, UPDATE, DELETE

leveraging the CLI tools available on a jailbroken device, such as sqlite3, users can execute SQL statements like INSERT, UPDATE, and DELETE directly on the SQLite databases with SSH connection to the mobile phone (Figure 19). This allows for more granular control over the data stored in various applications and system components. By modifying data, a criminal can attempt to create a false alibi by tampering with timestamps, location information, or communication records [13].

This can create a misleading digital trail that may support their fabricated alibi and complicate the investigation process. There are 2 approaches for doing this: direct or off-device [2]. In the off-device approach databases are extracted to a computer, manipulated and moved back to the file system. In the direct approach data is manipulated on the device with an SSH communication channel to a computer. The following methods will be presented to illustrate how data manipulation and the creation of false alibis can be achieved (Figure 20). This could also be automated by using a scripted program as studied by Albano et al [13]. When users utilize iMessages or FaceTime with end-to-end encryption, the records of these communications are not

```
sqlite> .open CallHistory.storedata
sqlite> .table
ZCALLDBPROPERTIES            Z_METADATA
ZCALLRECORD                  Z_MODELCACHE
ZHANDLE                      Z_PRIMARYKEY
Z_2REMOTEPARTICIPANTHANDLES
sqlite> SELECT ZLOCATION FROM ZCALLRECORD;
Estonia
Estonia
Estonia
Estonia
Estonia
Estonia
Estonia
Estonia
Estonia
sqlite> UPDATE ZCALLRECORD SET ZLOCATION = 'FINLAND' WHERE Z_PK = 4;
sqlite> SELECT ZLOCATION FROM ZCALLRECORD;
Estonia
Estonia
Estonia
FINLAND
Estonia
Estonia
Estonia
Estonia
Estonia
sqlite>
```

Figure 20 - Database record update

included in the network records provided by the network service provider. However, since the mobile network service providers offer 4G or 5G services used by mobile devices during these activities, they can track and record data usage and location information.

The following example demonstrates how easily data manipulation can be performed. A database CallHistory.storedata is opened. The aim is to manipulate the ZCALLRECORD of a data record where Z_PK is 4 from "Estonia" to "FINLAND". The manipulation in the example was intentionally presented in capslock for better visual representation (Figure 21). Its purpose is to illustrate how effortlessly data manipulations can be carried out, and when nothing appears suspicious or out of the ordinary, it can be perceived as accurate and truthful data. In certain situations, fabricating a false digital alibi may seem less suspicious at first than when the user has performed a factory reset on their device or intentionally destroyed their mobile or SSD storage.

	Z_PK	ZDATE	ZLOCATION	ZADDRESS
1	1	2023-04-18 11:53:13	Estonia	+37253408471
2	2	2023-04-26 05:41:42	Estonia	+37253408471
3	3	2023-04-26 05:41:47	Estonia	+37253408471
4	4	2023-04-26 05:41:53	FINLAND	+37253408471
5	5	2023-04-26 05:42:13	Estonia	+37253408471
6	6	2023-04-26 05:42:44	Estonia	+37253408471
7	7	2023-04-26 05:42:55	Estonia	+37253408471
8	8	2023-04-26 05:43:04	Estonia	+37253408471
9	9	2023-04-26 05:43:17	Estonia	+37253408471

Figure 21 - manipulated data viewed from DB Browser

7.2 Manipulating database behaviour

It has been covered previously how database pragma settings and triggers affect database behaviour. This paper has emphasized the need for a deeper understanding of iOS versions and their functionality. When the database has been manipulated to behave in ways it is not intended to, it becomes necessary for investigators to carefully analyze both the behaviour and the data. In real-life scenarios, such manipulation would lead to the dismissal of the evidence once manipulation has become apparent. In this section, we will perform three experiments to investigate different aspects of database manipulation. These experiments include selectively storing chosen content in the database, utilizing pragma commands for specific purposes, and creating triggers to control database behaviour.

7.2.1 Manipulating database tables

Now, the manipulation of database tables will be explored by deleting a specific table within the database. This action guarantees that any subsequent records created will not be stored, and the corresponding data will be dropped (Figure 22) effectively corrupting the database. A command "DROP table ZCALLRECORD;" removes the database table within CallHistory.storedata.

```
iPhone:/var/mobile/Library/CallHistoryDB root# sqlite3 CallHistory.storedata
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .table
ZCALLDBPROPERTIES            Z_METADATA
ZCALLRECORD                  Z_MODELCACHE
ZHANDLE                      Z_PRIMARYKEY
Z_2REMOTEPARTICIPANTHANDLES
sqlite> DROP TABLE ZCALLRECORD;
sqlite> .table
ZCALLDBPROPERTIES            Z_METADATA
ZHANDLE                      Z_MODELCACHE
Z_2REMOTEPARTICIPANTHANDLES  Z_PRIMARYKEY
sqlite>
```

Figure 22 - DROP table ZCALLRECORD

This leads to an interesting situation where the -wal file recognises a record being made, the file modification timestamp gets updated, but neither records a phone number nor creates an additional row with a new Z_PK number. Next, a hex dump of CallHistory.storedata-wal database was inspected, and also no records of this phone number were stored. However, calls can still be made and received, and unlike in "ghost messaging" explored later in this paper, these calls cannot be viewed from the GUI. Thus, presumably not even stored in RAM memory. Yet, interactionC.db database does record this record this call. At this particular moment, two calls were placed to both "Mary Jane" and "Jane Doe" (Figure 23). A reboot will not recreate the table ZCALLRECORD within the database.

ZLASTOUTGOINGRECIPIENTDATE	ZDISPLAYNAME
Filter	Filter
NULL	apple
2023-04-24 11:18:23	Mary Jane
2023-04-24 11:08:32	Jane Doe

Figure 23 - interactionC.db

Next, it will be analysed whether forensic tools can detect any call records. Two different tools were used, the iPhone Backup Extractor and Magnet Axiom, and neither of them were able to recover any call records originating from ZCALLRECORD table. Magnet Axiom is unable to recreate the data records from other databases like the interactionC.db. The database as it is seen, and it is lacking the table ZCALLRECORD.

7.2.2 PRAGMA commands

Pragma settings must be configured at the time of database creation. Only one pragma setting can be changed later in the database: changing auto_vacuum from incremental to full, as vacuum pragma recreates a new duplicate file. However, any valid pragma command can be made through the command line interface. Deleted records can be recovered as long as they remain within the WAL file or in the unallocated space of the database. Through the sqlite3 command line program the WAL file can be force

checkpointed to commit the WAL file to the main database file and then be truncated to minimise the size.

sms.db	4.1 KB	Today, 12:24
sms.db-shm	32.8 KB	Today, 12:24
sms.db-wal	267.8 KB	Today, 12:24

Figure 24 - sms.db before force commit

An sms.db database was chosen for this experiment as it has auto vacuum turned to "none", which disables the automatic vacuuming feature. This means that the database will not automatically reclaim unused space or rearrange data to reduce file size. As a result, the database file may grow in size over time. Any deleted records could potentially remain in the database for a very long time before they get overwritten (Figure 24).

```
iPhone:/var/mobile/Library/SMS root# sqlite3 sms.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> PRAGMA wal_checkpoint(TRUNCATE);
0|0|0
sqlite> .quit
iPhone:/var/mobile/Library/SMS root#
```

Figure 25 - PRAGMA wal_checkpoint(TRUNCATE) on sms.db

Under normal circumstances, based on the sms.db parameters, the -wal file should never get to a size of 0 bytes. Data within the -wal file will be overwritten by data already committed. This is a very small indication in a very complex system, most likely not a very reliable detection method.

The command "PRAGMA wal_checkpoint(TRUNCATE);" forces a checkpoint event on the database and commits the -wal file into the main database (Figure 25). The truncate command will reduce the file size; since all records from the -wal file were committed the size of the -wal file will get to 0 bytes (Figure 26).

sms.db	245.8 KB	Today, 13:15
sms.db-shm	32.8 KB	Today, 12:24
sms.db-wal	0 B	Today, 13:15

Figure 26 - sms.db after wal_checkpoint(TRUNCATE)

7.3 Deleting databases - "Ghost messaging"

Just as it is possible to make changes to the data records in a database, it is possible to delete or change files. Deleting a database involves removing the database file and any related journal files or directories, which can be accomplished without affecting the functioning of the underlying operating system. In most cases, a new database file together with its journal files is recreated at the time when the first data record is recorded in the database. This removes all of the data related to that application. The unique Z_PK primary key number starts again from "1", and the WAL file will contain only the most recent record.

However, it was observed that not every database will recreate the file at first data write and will require a reboot to recreate the database files. An example of such a database is sms.db, which is responsible for storing SMS/iMessage records and data from certain third-party messaging applications like WhatsApp. Until a reboot recreates the database files messages can be sent and received, but no data record gets recorded. When there is no existing database to store the data, it is simply dropped or discarded.

Figure 27 - "Ghost messaging"

Accessing the device's graphical user interface (GUI) can recent messages be viewed as they are temporarily retrieved and stored by the device's RAM memory (Figure 27).

However, these records will no longer be visible once the device is rebooted. A reboot will recreate a new database together with its journal files, and the database will contain no data records. The peculiarity of this behaviour most likely lies within the application database handler. The detailed explanation for this behaviour is beyond the scope of this thesis, as it focuses solely on observing and documenting the observed behaviour. This scenario illustrates that a form of "ghost messaging" is possible.

Next, it will be examined how forensic tools perceive this manipulated data. The anticipated outcome is that these messages will remain undetected. Forensic tools typically search for the presence of a database, and in cases where no database is found, they cannot provide any information. For the forensic investigator, the absence of an expected database like sms.db clearly indicates database manipulation, but it is the recovery of this evidence that they are most concerned about.

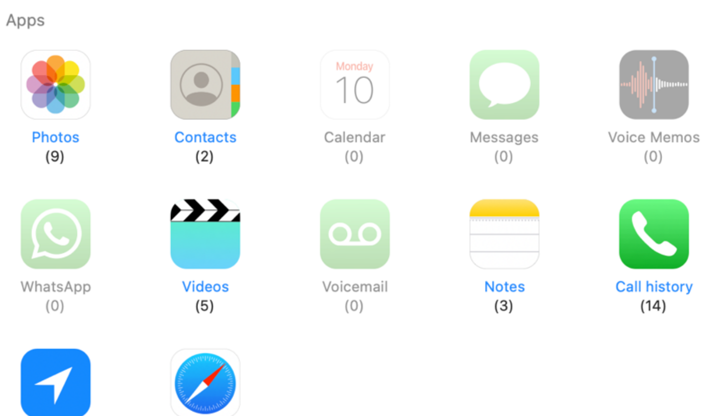

Figure 28 - no messages detected by iPhone Backup Extractor

First, iPhone Backup Extractor is used to make a backup of the device. Nevertheless, the iPhone Backup Extractor fails to identify the sms.db database, resulting in the absence of any retrieved messages (Figure 28). Furthermore, in the Expert mode, no SMS folder is generated at all. Next, Magnet Axiom is used, but it results in an identical result, no recovered artefacts (Figure 29). Therefore, it can be concluded that "ghost messages truly remain undetected by forensic tools. Until these messages remain in the RAM memory, only manual extraction of the live device is possible to gain access to this data; once the device is rebooted, this data is permanently lost.

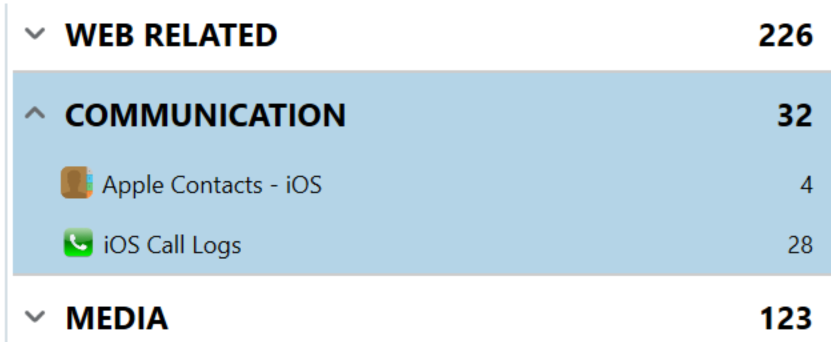

Figure 29 - no messages detected by Magnet Axiom

7.4 Creating database triggers

Database triggers are special database objects that are associated with tables or views and are executed automatically in response to certain events or actions. They allow you to define custom logic that is automatically executed when specific database operations occur. Database triggers can be made to drop saved records, manipulate them automatically for example every timestamp, add new records and so on. It would be possible to keep the mobile system time in the correct time, but for example, every sms sent or received will be timestamped a few hours earlier. In below is created a scenario where all deleted records ROWID, text, account and date will be saved to a new table called saved_data.

```
sqlite> CREATE TABLE saved_data (ROWID, text, account, date);
sqlite> .table
_SqliteDatabaseProperties  message
attachment                 message_attachment_join
chat                       message_processing_task
chat_handle_join           saved_data
chat_message_join          sync_deleted_attachments
deleted_messages           sync_deleted_chats
handle                     sync_deleted_messages
kvtable
sqlite> CREATE TRIGGER save_data AFTER DELETE ON saved_data
   ...> BEGIN INSERT into message (ROWID, text, account, date)
   ...> VALUES (OLD.ROWID, OLD.text, OLD.account, OLD.date);
   ...> END;
sqlite> .quit
```

Figure 30 - database trigger

7.5 Detecting through bash history

Any command given through the Bash command line interface is recorded in Bash history. The Bash history is a log of user commands [21]. By default, there are no commands stored in Bash history on iOS devices. While it may be a great way of detecting possible manipulations, this approach has many weaknesses.

- It records only commands given through the command line. Any change made through other programs like Secure File Transfer Protocol (SFTP) tools will not be recorded in Bash history. This includes whether the change is made in the command line but through another program like SQLite.
- For a successful detection, it is required to have an unaltered log file. Bash history in a jailbroken phone is easy to access through the command line and to clear history.
- The commands are recorded without timestamps by default; having commands without timestamps can give an idea of what has been done but lack contextual evidence as to "when" it was done. With a simple command the Bash history can be made to start recording commands with timestamps, but this will have little value in forensic investigations after the mobile device has been seized.

Having commands stored in iOS bash history in itself would not be a crime, some users who have decided to jailbreak their device for advanced customisation may incorporate iPhones command line. However, it could be a hint that the user has more technical knowledge and therefore potential to manipulate data. Analysing Bash history in computers may yield more meaningful results as the command line interface is available by default, but the same weaknesses remain.

Below is the presented a Bash history (Figure 31) from the actions shown in Figure 20. It can be seen that a CallHistoryDB directory has been entered, the user has started sqlite3 session and lastly the user has obsereved the Bash history. There are no timestamps or history of SQLite commands. While iOS and Mac OS X are both based on the same Unix system, iOS has considerably less packages installed

```
iPhone:/var/mobile/Library/CallHistoryDB root# history
    1  cd mobile/Library/CallHistoryDB
    2  sqlite3
    3  history
```

Figure 31 - bash history

7.5.1 Detecting Jailbreak

Based on the conducted experiments, the significance of detecting prior jailbreaking on the mobile device cannot be overstated. From previously jailbroken devices is easier to acquire data but most jailbreaks on iOS currently available are semi-tethered or semi-untethered jailbreaks [3]. This means that after each reboot, the device must be re-jailbroken for continued root rights. The prevalence of this happening is very high. Further, a user could intentionally attempt to remove traces of jailbreak. Based on the conducted experiments, the significance of detecting prior jailbreaking on the mobile device cannot be overstated. The determination of whether the mobile device has been jailbroken can significantly impact the reliability of the data [13]. It is important to consider that a user who is accustomed to a jailbroken iPhone possesses a greater familiarity and deeper knowledge of iOS [3]. However, in order to gain access to the data in the complete file system, the forensic examiner may need to perform their own jailbreak, which could potentially overwrite any traces of prior jailbreaks. Several methods still exist for detecting whether a mobile device has been jailbroken. These techniques can help forensic examiners identify signs of jailbreaking and assess the integrity of the device's data. The presence of unauthorized files serves as a distinct indicator of a jailbroken device (Figure 32). Identifying such files is a telltale sign that the device's operating system has been modified, allowing for the installation of applications or system alterations beyond the restrictions imposed by the manufacturer.

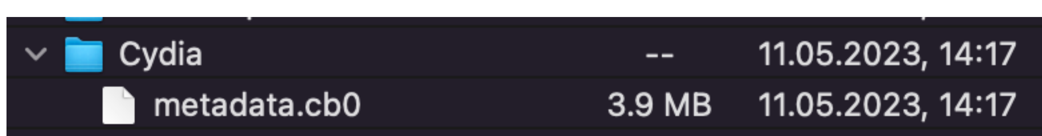

Figure 32 - Cydia, an unauthorised application

Unauthorised files may include custom system binaries, third-party applications not available through official app stores, or modifications to system settings. Detecting and analyzing these unauthorised files is crucial in determining whether a device has undergone jailbreaking. In var/ directory can be identified the checkra1n.dmg mountable disk image file (Figure 33).

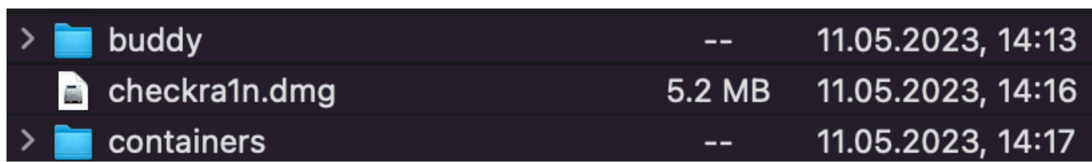

Figure 33 - checkra1n.dmg

8 Summary

The challenges addressed in this paper show how in relatively simple ways the data or databases can be manipulated ja while detection may be possible it illustrates the need for greater knowledge and understanding of iOS systems. Databases have different settings and parameters that affect the way data is stored and handled within the database. This research sheds light on lesser-explored methods of data manipulation that have received limited attention in the past. It explores concepts of not only deleting data to hide traces but ways to avoid data storage in databases altogether. It highlights how data or databases can be manipulated even with relatively simple techniques with no specialised tools, posing significant challenges in digital forensic investigations. While detecting potential instances of manipulation may be possible, recovering deleted data is a more complex and challenging task. Deleted data may not be easily recoverable, as it can be overwritten or fragmented, making it challenging to reconstruct a complete picture of the original information. In some cases, exploring previous backups may be beneficial in hopes of recovering some lost data.

Data obtained from mobile phones play a crucial role in digital forensic investigations, serving as a significant component of the overall evidence. The wealth of available

smartphone data offers a well-defined snapshot of user activities and events. By examining this data, forensic investigators can gain valuable insights into a user's interactions, communications, internet browsing history, location information, and other relevant information. Such data acts as a valuable resource for reconstructing events, establishing timelines, and supporting the investigation process. However, as this study presents, any data gathered from mobile devices should be critically verified and analysed. Evidence can be cross-referenced to one another for additional reliability. By comparing and correlating multiple pieces of evidence, investigators can strengthen their confidence in the findings and conclusions drawn from the investigation. Remote traces often hold greater probative value compared to local traces in the digital forensic investigations [13]. For instance, call and location data obtained directly from a mobile service provider is considered more reliable and verifiable than data derived solely from the device itself. This research was limited to only full file system extraction, but a more thorough physical acquisition could potentially recover more deleted or manipulated records. This research puts data records gathered from jailbroken mobile devices under serious question. Chapter 7 shows how data can be manipulated in ways not always detectable by forensic tools. Digital forensic investigations typically do not perform investigations on live raw data and unless a reasonable amount of suspicion is caused such data manipulations may remain undetected. The case is further complicated by the amount of knowledge required to know in detail how different applications store data in databases. Every iOS update or new iPhone model could be vastly different from the previous.

9 Conclusions

The paper brings attention to the intricate nature of data manipulation and underscores the inherent difficulties of recovering accurate and unaltered data. It sheds light on the complex processes and techniques that malicious actors could employ to manipulate data, potentially to indistinguishable extents. It underscores the need for robust forensic methodologies, advanced tools, and expert knowledge to effectively investigate and analyze digital evidence in the face of these challenges. Jailbroken devices, however, pose unique challenges to the irrefutability of digital evidence. It may determine either conviction or acquittal of a suspect.

Future work can build on this research by establishing more connections to database behaviour and the identification of manipulated data. These experiments should be conducted on other versions of iOS and on Android, especially on the "ghost messaging" capabilities. The findings presented in this research hold potential value for forensic tool creators in the development of automated tests to detect manipulation traces. By understanding the various methods of data manipulation explored in the study, forensic tool creators can leverage this knowledge to design algorithms and techniques that can automatically identify and flag potential manipulation indicators.

10 References

[1] H. Pieterse, M. Olivier and R. van Heerden, "Evaluating the Authenticity of Smartphone Evidence," 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-67208-3_3. [Accessed 17 04 2023].

[2] H. Pieterse, M. Olivier and R. van Heerden, "Detecting Manipulated Smartphone Data on Android and iOS Devices," 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-11407-7_7. [Accessed 26 04 2023].

[3] Y.-T. Chang, K.-C. Teng, Y.-C. Tso and S.-J. W. Wang, "Jailbroken iPhone Forensics for the Investigations and Controversy to Digital Evidence," Department of Information Management, Central Police University, 2015.

[4] A. Al-Dhaqm, S. Abd Razak, R. A. Ikuesan, V. R. Kebande and K. Siddique, "A Review of Mobile Forensic Investigation Process Models," 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9160916. [Accessed 15 03 2023].

[5] CEN Workshop Agreement, "CWA 17865:2022; Requirements and Guidelines for a complete end-to-end mobile forensic investigation chain," CEN, 2022.

[6] J. Gruber, C. J. Hargreaves and F. C. Freiling, "Contamination of digital evidence: Understanding an underexposed risk," 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281723000021. [Accessed 10 05 2023].

[7] S. S. Shimmi, G. Dorai, U. Karabiyik and S. Aggarwal, "Analysis of iOS SQLIte Schema Evolution for Updating Forensic Data Extraction Tools," 2020.

[8] R. Tamma, H. Skulkin, H. Mahalik and S. Bommisett, Practical Mobile Forensics - Fourth Edition, Packt, 2020.

[9] S. Garg and N. Baliyan, "Comparative analysis of Android and iOS from Security viewpoint," Computer Science Review, 2021.

[10] Apple inc, "Developer Documentation," [Online]. Available: https://developer.apple.com/documentation/technologies. [Accessed 01 03 2023].

[11] Apple inc., “Support Documentation - iPhone,” [Online]. Available: https://support.apple.com/iphone. [Accessed 30 03 2023].

[12] Z. A. M. Burgos, “Jailbreak Vulnerability & Mobile Security Updates,” 2018.

[13] P. Albano, A. Castiglione, G. Cattaneo, G. De Maio and A. De Santis, “On the construction of a False Digital Alibi on the Android OS,” in *Third International Conference on Intelligent Networking and Collaborative Systems*, Fisciano, Italy, 2011.

[14] A. De Santis, A. Castiglione, G. Cattaneo, G. De Maio and Ianulardo, Springer, 2011.

[15] C. Meng and H. Baier, “bring2lite: A Structural Concept and Tool for Forensic Data Analysis and Recovery of Deleted SQLite Records,” Elsavier, Germany, 2019.

[16] Y. Liu, M. Xu, J. Xu, N. Zheng and X. Lin, “SQLite Forensic Analysis Based on WAL,” 2017.

[17] SQLite, “Database File Format,” [Online]. Available: https://www.sqlite.org/fileformat.html. [Accessed 02 05 2023].

[18] SQLite, “Write-Ahead Logging,” 2018. [Online]. Available: https://www.sqlite.org/wal.html. [Accessed 04 04 2023].

[19] S. Punja and I. Whiffin, “Missing SQLite Record Analysis,” 2021.

[20] SQLite, “PRAGMA Statements,” [Online]. Available: https://www.sqlite.org/pragma.html. [Accessed 13 05 2023].

[21] J. Hance and J. Straub, “Use of Bash History Novelty Detection for Identification of Similar Source Attack Generation,” *IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom),* 2020.

[22] S. Nemetz, S. Schmitt and F. Freiling, “A standardized corpus for SQLite database forensics,” Elsavier, 2018.