2026 EDITION
Latest techniques, tools & security updates

Practical Mobile Forensics

Forensically investigate and analyze modern iOS and Android devices

Covers iOS 26 and Android 17





Fifth Edition

ROHIT TAMMA

‹packt›

Practical Mobile Forensics

Fifth Edition

Forensically investigate and analyze modern iOS and Android devices

Rohit Tamma

‹packt›

Practical Mobile Forensics

Fifth Edition

Copyright © 2026 Packt Publishing
All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Vijin Boricha
Portfolio Manager: Prachi Sawant
Program Manager: Ankita Thakur
Content Engineer: Swathi Ajayakumar
Technical Editor: Nithik Cheruvakodan
Indexer: Tejal Soni
Proofreader: Swathi Ajayakumar
Production Designer: Salma Patel
Growth Lead: Ankita Thakur

First published: July 2014
Second edition: May 2016
Third edition: January 2018
Fourth edition: April 2020
Fifth edition: August 2026

Production reference: 1280826
Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80610-777-3
www.packtpub.com

Contributors

About the author

Rohit Tamma is a cybersecurity expert with a deep focus on leading teams that secure enterprises from cybersecurity attacks. With over 17 years in the industry, he has extensive technical leadership experience in areas like security operations, penetration testing, cloud security and mobile forensics. He currently works at Google as a Security Engineering Manager.

About the reviewer

Michael Yasumoto is a senior forensic examiner at Deadbolt Forensics, a digital forensics consulting firm located in Portland, Oregon. He has been practicing digital forensics for over 10 years and has taught hundreds of examiners as an instructor for AccessData and now H-11 Digital Forensics. Michael holds a master's degree in computer science from George Washington University and his forensic certifications include: CFCE, CAWFE, ICMDE, GCFE, GCFA, GNFA, GREM, GIME, GASF, CCME, X-PERT.

Table of Contents

Preface xv

Free benefits with your book xx

Part 1: iOS Forensics 1

Chapter 1: Introduction to Mobile Forensics 3

Mobile devices as sources of digital evidence 4

Forensic principles and evidence integrity 5

Mobile forensic investigation process 6

Applying the mobile forensics workflow • 8

Evidence intake and legal authority • 8

Identification and preservation • 9

Preparation • 9

Isolation • 10

Acquisition • 10

Verification • 11

Examination and analysis • 11

Documentation and reporting • 11

Archiving • 12

Mobile platforms and forensic considerations 12

The forensic pyramid classification system • 13

Manual extraction • 14

Logical extraction • 14

Hex dump / physical extraction • 14

Chip-off/ISP • 15

Micro read • 16

Types of evidence extracted from mobile devices • 16

Examining and analyzing extracted data • 17

Legal rules and handling of evidence • 18

Best practices for mobile forensic investigations 20

Securing evidence • 20
Preserving evidence • 21
Documenting the evidence and changes • 21
Reporting • 22
Challenges in modern mobile investigations 23
Summary 26
References 27

Chapter 2: iOS Architecture, Security, and Filesystem Overview 29

Introduction to iOS ecosystem 30
iOS architecture layers 32
Core OS layer • 32
Core services layer • 33
Media layer • 33
Cocoa touch layer • 33
iOS security model 34
Hardware security • 34
Secure Enclave processor (SEP) • 34
UID and GID keys • 34
Memory integrity enforcement • 34
System security • 35
Secure Boot chain • 35
Software updates • 35
Kernel integrity • 36
Application security • 37
App sandboxing • 37
Code signing • 37
Entitlements • 38
Keychain and iCloud security 38
Access groups and access control • 39
iCloud Keychain and end-to-end encryption • 39
Forensic relevance and extraction challenges • 40
Apple file system (APFS) 41
Key differences from HFS+ • 43
Important directories in iOS • 43

Snapshots and their forensic implications • 44
Forensic considerations with APFS • 45
Impact of modern security enhancements on forensics 45
File system encryption vs. full disk encryption • 46
Secure Enclave processor and passcode resistance • 46
Secure Boot chain and imaging limitations • 47
Jailbreaking and its forensic implications 48
Summary 50

Chapter 3: Data Acquisition from iOS Devices 53

Operating modes of iOS devices 54
Normal mode • 55
Recovery mode • 56
Entering recovery mode • 58
Exiting recovery mode • 58
DFU mode • 59
Entering DFU mode • 59
Verifying DFU mode • 60
iOS-specific legal and forensic considerations 61
Passcodes, encryption, and the Secure Enclave • 61
Jailbreaking, exploits, and legal boundaries • 61
Legal authority and warrants in the iOS context • 62
Documentation and chain of custody • 62
iOS isolation techniques to prevent remote wipe and data modification 62
Remote wipe and activation lock • 63
Effective isolation strategies • 63
Power and battery considerations • 64
iOS logical acquisition 64
Prerequisites and limitations • 64
Tools for logical acquisition • 65
Forensic value and best practices • 66
Logical acquisition in practice • 66
Practical walkthrough: Logical acquisition using a commercial forensic tool • 67
iOS filesystem acquisition 70
What makes filesystem acquisition different and valuable • 71

Agent-based acquisition • 72
Jailbreak-assisted acquisition • 73
Practical walkthrough: Agent-based filesystem acquisition • 75
Practical walkthrough: Jailbreak-assisted filesystem acquisition • 76
iOS physical acquisition 79
iOS diagnostic and unified log acquisition 80
Understanding unified logs and sysdiagnose • 80
Forensic value of unified logs • 81
Collecting diagnostic and unified logs • 81
Manual extraction pathways (without FFS) • 82
Acquiring full unified log archives (~30 days) • 82
Limitations and best practices • 83
Decision framework for selecting iOS acquisition methods 83
Step 1: Starting with the device model and processor generation • 83
Step 2: Considering the iOS version • 84
Step 3: Locked or unlocked state of the device • 84
Step 4: Matching legal authority with acquisition depth • 84
Step 5: Mapping acquisition options • 84
Step 6: Balancing completeness and risk • 85
Summary 85

Chapter 4: Data Acquisition from iOS Backups 87

Introduction to iOS backups 88
The role of backups in forensic investigations • 88
Backups vs physical and logical acquisition • 89
Local vs. iCloud backups • 90
Local backups using Finder, Apple Devices, and iTunes 92
Creating local backups • 93
Locating local backups • 94
Standard backup metadata files • 95
Backup contents • 95
Encrypted vs. unencrypted backups • 96
Extracting backups on macOS • 96
Forensic best practices • 97
Practical walkthrough: Extracting an iOS backup using idevicebackup2 (open-source) • 98

Practical walkthrough: Analyzing an iOS backup using iBackup Viewer • 101
iCloud backups 105
Understanding how iCloud backups work • 106
Types of data in iCloud backups • 106
Accessing iCloud backups • 107
Impact of ADP • 108
Practical walkthrough: Acquiring an iCloud backup using Magnet AXIOM • 109
Summary 115

Chapter 5: iOS Data Analysis and Recovery 117

Foundations of iOS data interpretation 118
Parsed results vs. actual evidence • 118
Tool output and raw data validation • 119
Filesystem artifacts vs. backup-derived artifacts • 119
Understanding iOS time and metadata 120
Unix timestamps • 121
Mac absolute time • 122
WebKit and Chrome time • 123
Time zones, device settings, and interpretation errors • 123
SQLite databases in iOS: Structure and forensic relevance 124
Why iOS uses SQLite everywhere • 124
Examining SQLite databases in practice • 125
Working with SQLite databases • 126
Connecting to a database • 127
Exploring SQLite key commands • 127
How SQLite evidence is parsed and why tools can differ • 130
Filesystem extraction vs. backup-based SQLite evidence • 130
Locating SQLite databases in iOS evidence 130
Locating SQLite databases in filesystem extractions • 131
Locating SQLite files from iOS backups • 132
Core artifacts: Analyzing key iOS database files 135
Contacts artifact: AddressBook.sqlitedb • 136
Call history artifact: CallHistory.storedata • 140
Messages artifact: Sms.db • 143
Calendar events artifact: Calendar.sqlitedb • 144

Notes artifact: NoteStore.sqlite • 147
Safari bookmarks, history, and tab artifacts • 148
Device interaction and behavioral intelligence artifacts • 150
User's phone numbers • 151
Property lists (plist) and their forensic significance • 152
Applying artifact analysis beyond core databases • 158
Deleted data recovery and limitations 158
SQLite freelist pages • 159
Write-ahead log (WAL) files • 161
Unallocated space in file-based acquisitions • 163
Recently deleted vs. soft-delete artifacts • 163
Impact of modern iOS security • 163
Practical expectations for examiners • 164
Summary 164

Chapter 6: iOS Forensic Tools and Automation 167

iOS forensics tooling landscape 168
Functional roles of iOS forensic tools • 168
Commercial forensic platforms • 169
Open-source and manual analysis tools • 170
Combining automated and manual analysis • 170
Tool validation and evidentiary reliability 171
Cross-tool validation • 171
Source artifacts vs. parsed output • 172
Timestamp verification and chronological integrity • 172
Schema interpretation and record count confirmation • 173
Deleted data claims and residual artifact validation • 173
Version compatibility and parser limitations • 174
Documentation and defensibility • 174
Role and limits of automation in iOS forensics 175
What should be automated? • 175
What should remain examiner-driven? • 176
Automation can scale errors too • 176
Maintaining provenance during automation • 176
Human-guided automation • 177

Working with commercial iOS forensic platforms 177
Importing and preparing evidence • 177
Automated artifact parsing • 179
Searching and filtering artifacts • 180
Timeline analysis and activity reconstruction • 180
Validating critical findings • 181
Exporting evidence and generating reports • 181
How commercial platforms differ in practice • 182
Automation techniques for iOS artifact analysis 182
Automating SQLite artifact extraction • 183
Automating artifact extraction across multiple backups • 184
Automated timeline construction and correlation • 186
Automation best practices • 189
Summary 190

Part 2: Android Forensics 191

Chapter 7: Understanding Android Architecture 193

Evolution of Android and its forensic impact 194
Android architecture overview 196
Linux kernel layer • 198
Hardware abstraction layer (HAL) • 198
Native libraries and Android Runtime (ART) • 198
Application framework • 199
System applications and user applications • 199
Application execution environment 200
Dalvik vs ART (evolution and current state) • 200
Application storage model • 202
Android security model 203
Linux permission model and UID/GID • 203
Application sandbox enforcement • 203
Permission model (install-time vs. runtime permissions) • 204
Securing inter-process communication (IPC) • 205
Application signing and trust model • 205
Security-Enhanced Linux (SELinux) • 206

Encryption in Android **207**
Full disk encryption (FDE) • 207
File-based encryption (FBE) • 207
Device encrypted (DE) storage • 208
Credential encrypted (CE) storage • 208
Forensic implications of Android encryption • 209
Hardware-backed security mechanisms **209**
Android Keystore • 209
Trusted execution environment (TEE) • 210
StrongBox secure element • 211
Verified Boot and system integrity **212**
Chain of trust • 212
Android Verified Boot (AVB) • 212
Bootloader locking and forensic impact • 213
Android storage and filesystem layout **213**
General partition structure • 213
Dynamic partitions and A/B updates • 214
Application data and user artifacts • 215
Emulated storage (/sdcard) • 215
Scoped storage restrictions • 215
Filesystems in Android **216**
EXT4 filesystem • 218
F2FS filesystem • 218
Emulated and virtual filesystems • 219
Journaling, WAL, and forensic considerations • 219
Legacy filesystems • 219
From Android architecture to acquisition strategy **220**
Summary **221**

Chapter 8: Android Forensic Setup and Pre-Data Extraction Techniques **223**

Introduction to pre-extraction workflows **224**
Forensic workstation and tooling setup **225**
Installing Android platform tools • 225
Verifying ADB connectivity • 226

Understanding ADB device states • 227
Drivers and system considerations • 229
Pre-connection validation • 229
Environment readiness checklist • 230
Handling and preparing the device for examination 230
Initial device assessment • 230
Network isolation • 231
Maintaining device state • 231
Enabling USB debugging • 232
Preserving evidence integrity • 233
Screen lock and access considerations 234
Impact on data accessibility • 234
Unlocked vs. locked state • 235
Practical and forensic considerations during acquisition • 236
Access strategies and practical approaches 237
Scenario 1: Device locked and no prior access • 238
Scenario 2: Device locked but ADB previously authorized • 239
Scenario 3: Device unlocked at time of seizure • 239
Scenario 4: Device powered off • 240
Scenario 5: Device unlocked but restricted access • 240
Rooting and privilege escalation 240
Understanding the rooting process • 241
Rooting in modern Android environments • 243
Root access: ADB shell • 245
Limitations in modern Android • 246
Forensic risks and considerations • 247
Practical and forensic perspectives • 247
Working with ADB shell: Essential Linux commands..... 248
Validation and pre-extraction checklist 250
Final pre-extraction check • 252
Summary 252

Chapter 9: Unlock Your Exclusive Benefits 255

Unlock this Book's Free Benefits in 3 Easy Steps..... 256

Other Books You May Enjoy **260**

Index **263**

Preface

Mobile forensics has evolved significantly with the rise of secure hardware, encrypted ecosystems, and cloud-first architectures. *Practical Mobile Forensics, Fifth Edition* explores the science of acquiring and analyzing data from mobile devices in a forensically sound manner, while addressing the challenges posed by modern operating systems and security controls.

This edition focuses on practical, real-world techniques for investigating mobile devices across contemporary platforms, including the latest versions of iOS and Android. The book covers both open-source and commercial forensic tools, guiding you through structured workflows for acquisition, analysis, and reporting. As mobile ecosystems become more restrictive, the book also examines platform-level limitations, encryption models, and practical approaches to navigate these constraints.

By the end of this book, you will have a strong, hands-on understanding of modern mobile forensics, enabling you to extract, analyze, and interpret data from mobile devices, cloud ecosystems, and backups using reliable and defensible methods. This book will also equip you to apply advanced forensic techniques to recover deleted data from mobile devices, as well as understand the limitations of modern mobile forensics and approaches to navigate them.

Who this book is for

This book is designed for digital forensic practitioners and investigators looking to build foundational skills in mobile forensics across modern iOS and Android platforms. It is also valuable for security professionals, incident responders, and researchers interested in understanding mobile device internals, application data, and forensic artifact analysis. A foundational understanding of digital forensics and basic familiarity with operating systems will help readers get the most from this book, though prior forensic experience is not mandatory.

What this book covers

Chapter 1, *Introduction to Mobile Forensics*, introduces you to the concepts of mobile forensics, the core values and the challenges involved. This chapter also provides an overview of practical approaches and best practices involved in performing mobile forensics.

Chapter 2, *iOS Architecture, Security, and Filesystem Overview*, provides an insight into iOS forensics. You will learn about the filesystem layout, security features, and the way files are stored on an iOS device.

Chapter 3, *Data Acquisition from iOS Devices*, introduces tools that will aid in obtaining data from iOS devices to later forensically examine. Not all tools are created equally, so it's important to understand the best tools to get the job done properly.

Chapter 4, *Data Acquisition from iOS Backups*, discusses iOS device backup files in detail, including user, forensic, encrypted, and iCloud backup files, and the methods to conduct your forensic examination.

Chapter 5, *iOS Data Analysis and Recovery*, goes further into the forensic investigation by showing you how to analyze the data recovered from the backup files. Areas containing data of potential evidentiary value are explained in detail.

Chapter 6, *iOS Forensic Tools and Automation*, introduces commercial tools such as Elcomsoft iOS Forensic Toolkit, Cellebrite (UFED4PC, Touch, and Physical Analyzer), Oxygen Forensic Detective, AccessData MPE+, and many more, which are available for forensic acquisition and the analysis of iOS devices. This chapter details steps required to perform acquisitions and analysis of iOS devices.

Chapter 7, *Understanding Android Architecture*, covers the Android platform and how to perform forensics on Android devices.

Chapter 8, *Android Forensic Setup and Pre-Data Extraction Techniques*, explains the process of, and what to consider when, setting up a digital forensic examination environment.

To get the most out of this book

- A basic understanding of digital forensics concepts and practices, including evidence handling, forensic acquisition, data analysis, and maintaining evidence integrity.
- Familiarity with Windows or Linux systems, mobile devices, file systems, and basic command-line tools. Prior experience with mobile forensics is not required.
- A dedicated forensic workstation with sufficient storage for device images, backups, and extracted data.
- The required iOS and Android utilities, device drivers, forensic tools, and supporting software, as introduced in the relevant chapters.
- Access to test iOS and Android devices and suitable USB cables for completing the practical exercises.
- Administrator privileges on the forensic workstation to install and configure the tools required for acquisition and analysis.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user inputs. For example: "Extract meaningful artifacts from the `/private/var` subtree."

A block of code is set as follows:

```
SELECT
    summary,
    datetime(start_date + 978307200, 'unixepoch') AS start_time,
    datetime(end_date + 978307200, 'unixepoch') AS end_time,
    description
FROM CalendarItem
ORDER BY start_date DESC
LIMIT 50;
```

Any command-line input or output is written as follows:

```
adb shell cat /proc/mounts
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Navigate to the **Browse Data** tab in **DB4S** and select the **Files** table."

> **Note**
>
> Warnings or important notes appear like this.

> **Tip**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Share your thoughts

Once you've read *Practical Mobile Forensics*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



https://packt.link/r/1806107775

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Part 1

iOS Forensics

This section provides a comprehensive look into iOS device forensics, focusing on iPhones, iPads, and the modern iOS operating system. As Apple continues to advance security and privacy features such as Lockdown Mode, Secure Enclave, Passkeys, Rapid Security Responses, Memory Integrity Enforcement (MIE), Advanced Data Protection (ADP), and the Passwords app introduced in recent iOS releases including iOS 26, digital investigations involving iOS have become more challenging and specialized.

You will explore a wide range of forensic acquisition techniques that help you extract data from iOS devices. Beyond acquisition, emphasis is placed on analysis. You will learn how to interpret system and third-party artifacts such as messages, app caches, location data, photos, and deleted content. Modern tools are also discussed in depth, with comparisons of their strengths and limitations.

This section includes the following chapters:

- *Chapter 1, Introduction to Mobile Forensics*
- *Chapter 2, iOS Architecture, Security, and Filesystem Overview*
- *Chapter 3, Data Acquisition from iOS Devices*
- *Chapter 4, Data Acquisition from iOS Backups*
- *Chapter 5, iOS Data Analysis and Recovery*
- *Chapter 6, iOS Forensic Tools and Automation*

Whether you are working with a modern iPhone running iOS 26, an older device running or a jailbroken device with legacy software, this section will help you understand how to acquire, analyze, and report data from iOS devices in a way that is effective, reliable, and legally sound.

1

Introduction to Mobile Forensics

If someone were to pick up your phone right now, they could reconstruct much of your recent life: where you went, whom you contacted, what you purchased, and which services you used. This deeply personal record makes mobile devices important sources of evidence in criminal, civil, corporate, and counterintelligence investigations. Mobile devices have become silent witnesses to our lives, recording where we go, who we talk to, what we buy, and even how we feel. This deeply personal footprint makes smartphones and tablets not just essential to daily life, but also treasure troves of digital evidence during civil, criminal, and counterintelligence investigations.

This reality has driven the rapid evolution of **mobile device forensics**, a specialized branch of digital forensics dedicated to extracting, preserving, and analyzing data from mobile devices in ways that are both legally sound and technically reliable.

This book provides a practical, hands-on understanding of forensic techniques for the two dominant mobile platforms: **Android** and **iOS**. We'll explore a variety of methods to collect, examine, and present evidence from different devices, covering everything from foundational practices to more advanced approaches.

In this chapter, we will cover the following topics:

- Mobile devices as sources of digital evidence
- Forensic principles and evidence integrity
- Mobile forensic investigation process
- Mobile platforms and forensic considerations
- Best practices for mobile forensic investigations
- Challenges in modern mobile investigations

By the end of this chapter, you will be able to explain why mobile devices are important sources of digital evidence; describe the principal stages of a mobile forensic investigation; compare common acquisition approaches; identify major categories of mobile evidence; and discuss the technical, operational, and legal challenges that affect evidence preservation, analysis, and reporting.

Note

Download the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Mobile devices as sources of digital evidence

Mobile devices are now the primary means by which billions of people communicate, access online services, make payments, navigate, and create digital content. Their scale and centrality to everyday life make them increasingly common sources of evidence. Current adoption, connection, shipment, and usage figures should be cited from the relevant industry reports because these values change rapidly.

Global mobile data traffic has also exploded over the last few years. Ericsson's latest Mobility Report projects mobile data traffic is expected to grow by a factor of around 2.2 to reach 328 EB per month in 2031. That's a staggering compound annual growth rate (CAGR) of over 25%. This surge is fueled by video streaming, cloud apps, social platforms, internet of things (IoT) integrations, and more sophisticated mobile services.

Today's smartphones are effectively powerful computers in miniature form. They come equipped with multi-core processors, AI accelerators, vast storage capacities (often hundreds of gigabytes), and advanced sensors. Beyond basic communication like calling and texting, they serve as hubs for internet browsing, encrypted messaging, digital payments, navigation with precise GPS, photography and videography, cloud document editing, health monitoring, and even secure authentication for enterprises.

As mobile devices evolve, the volume and sensitivity of data stored on them has grown exponentially. These devices log everything: location histories, social interactions, financial transactions, private messages, work documents, health metrics, and more, turning them into highly personal digital diaries. Consequently, mobile phones have become critical sources of digital evidence in investigations across criminal, civil, corporate, and counterintelligence domains. It is now rare to conduct a digital forensics case without analyzing at least one

mobile device. From organized crime to insider threats and civil disputes, mobile forensics is often the key to uncovering timelines, associations, and intent.

Real-world examples highlight this importance. In the investigation of the 2022 University of Idaho murders, investigators relied heavily on mobile phone evidence to reconstruct the suspect's movements and activities before and after the crime. By analyzing historical location information, device movement patterns, communication records, and other digital artifacts, investigators were able to establish timelines, identify repeated visits to the area, and correlate mobile device activity with other forms of evidence. The case demonstrated how modern smartphones can provide detailed insights into an individual's behavior, movements, and associations, even when traditional witnesses or surveillance footage are limited.

Similarly, mobile device evidence continues to play a critical role in ransomware investigations, insider threat cases, corporate espionage incidents, and financial crime investigations. Even when communications are protected by end-to-end encryption, you can often recover valuable evidence from device artifacts, application data, metadata, cloud-synchronized information, location history, and account activity records. In many cases, seemingly insignificant artifacts become the key to reconstructing events and establishing intent.

The discipline of acquiring, preserving, examining, analyzing, and reporting digital evidence from mobile devices is known as **mobile forensics**. Digital evidence includes data stored, processed, transmitted, or received by an electronic device that may be relevant to an investigation. Because mobile devices can contain communications, location information, application activity, media, documents, and cloud-synchronized data, they are often important evidence sources in criminal, civil, corporate, and internal investigations.

The following sections introduce the principles, methods, and evolving challenges of mobile forensics.

Forensic principles and evidence integrity

Digital forensics is the application of forensic methods to data stored, processed, or transmitted by electronic systems. Its branches include computer, network, cloud, IoT, and mobile-device forensics. The objective is to identify, acquire, preserve, examine, analyze, and report relevant data through methods that produce reliable and reproducible results.

Mobile forensics applies these principles to smartphones, tablets, smartwatches, and other connected mobile devices. A central principle is forensic soundness: evidence must be handled using methods that preserve its integrity and produce results that can be independently reviewed and reproduced.

In conventional computer forensics, storage media such as hard drives can often be removed from a system and connected to a forensic workstation through a hardware write blocker. This prevents write commands from reaching the storage medium during acquisition. Mobile devices operate differently. Their storage is normally integrated into the device, and the operating system and security architecture control access to the underlying data. As a result, you usually cannot remove the storage and examine it through a conventional write blocker.

Instead, forensic tools may need to communicate with the device through operating-system or vendor interfaces such as the **media transfer protocol** (**MTP**), **picture transfer protocol** (**PTP**), **Android debug bridge** (**ADB**), Apple backup services, or proprietary forensic protocols. More advanced acquisition techniques may involve a forensic agent, custom recovery environment, privileged exploit, bootloader operation, or hardware-level procedure.

Because these methods interact with the device, they may change system logs, temporary files, application state, metadata, or other device information. Traditional write blocking is therefore not always possible in mobile forensics. Forensic soundness does not mean claiming that no change occurred. It means understanding and minimizing any changes, documenting the actions that caused them, and demonstrating that they did not compromise the reliability of the acquired evidence or your findings.

You should maintain detailed examination notes, acquisition logs, tool and version information, configuration settings, timestamps, hash values, and chain-of-custody records throughout the investigation. Proper documentation allows another examiner to review the methodology, reproduce relevant steps where practical, evaluate any changes introduced during acquisition, and assess the reliability of the conclusions.

Mobile forensic investigation process

A mobile forensic investigation generally progresses through several interconnected phases. Although the exact workflow varies according to the device, legal authority, investigative objectives, and available tools, the following phases provide a useful framework. *Figure 1.1* provides an overview of various phases involved in the mobile forensics process:

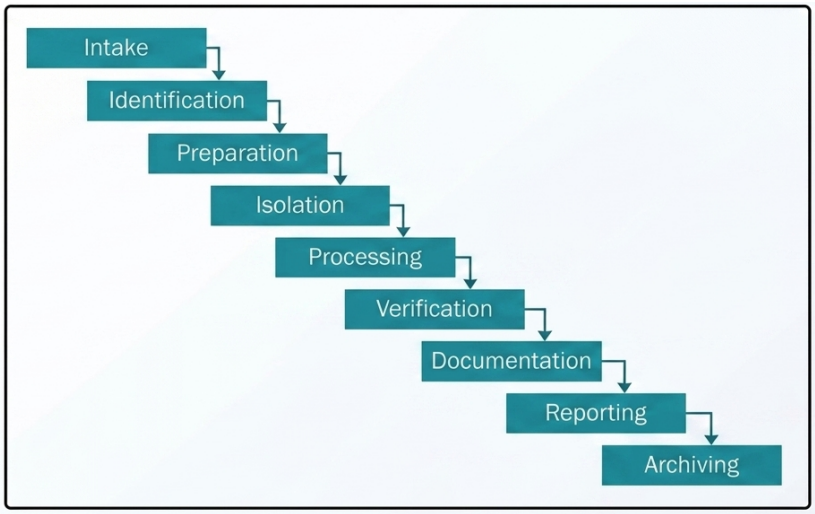


Figure 1.1: Mobile forensic process overview

- **Evidence intake and legal authority**: Establish the investigative objectives, record known case information, and confirm the authority under which the device and associated data may be examined.
- **Identification and Preservation**: Document the device, accessories, physical condition, power state, lock state, network status, and other characteristics that may affect preservation and acquisition.
- **Preparation**: Once the device has been secured, examiners search the device and develop an acquisition strategy. This typically involves identifying the make, model, operating system version, security patch level, storage configuration, and security mechanisms present on the device. They also assess potential risks such as encryption, screen locks, remote wipe capabilities, and acquisition-related data modification. Based on these findings, appropriate tools and techniques are selected for the examination.
- **Isolation**: Reduce the risk of remote commands, synchronization, incoming communications, or other activity that could alter evidence, while considering the consequences of interacting with or powering down the device.
- **Acquisition**: Collect relevant and authorized data using logical, backup-based, filesystem, physical, cloud, or other supported methods.
- **Verification**: Confirm that the acquisition completed as expected, document its contents and limitations, and generate cryptographic hashes to protect the integrity of acquired datasets and working copies.

- **Examination and analysis**: Identify, recover, validate, correlate, and interpret artifacts relevant to the investigative questions.
- **Documentation and reporting**: Maintain contemporaneous examination notes and produce a report explaining the methods, findings, limitations, and conclusions.
- **Archiving**: Preserve the acquired data, working materials, tool logs, notes, reports, hash values, and chain-of-custody documentation for future reviews.

These phases are not always performed in a strictly linear order. You may return to an earlier phase when new information is discovered, an acquisition method fails, or additional legal authority is required. The following example demonstrates how the workflow may be applied in practice.

Applying the mobile forensics workflow

The precise steps in a mobile forensic examination depend on the device, operating system, security configuration, legal authority, investigative objectives, and available acquisition methods. To demonstrate how the phases fit together, this section applies the workflow to a hypothetical corporate investigation:

A company suspects that a departing employee used a company-issued smartphone to obtain confidential product designs and disclose them to a competitor. You are asked to determine whether the employee accessed or photographed the designs, transferred them through a messaging application, uploaded them to cloud storage, or communicated them to an external party.

This example will be revisited throughout the workflow. At each phase, you must consider not only what information you want to recover, but also whether the proposed action is authorized, technically appropriate, and likely to preserve the integrity of the evidence.

Evidence intake and legal authority

You should begin by establishing the nature of the incident, the questions the examination must answer, and the scope of your work. Record the device's ownership, the employee's role, the suspected conduct, the relevant timeframe, and the categories of information believed to have been disclosed.

Before examining the device, you must confirm that you have the appropriate legal authority. Depending on the jurisdiction and circumstances, this may include a search warrant, documented consent, corporate authorization, an applicable employment policy, a contractual provision, or another recognized legal basis. You must keep the examination within the scope of that authority. For example, authorization to investigate the suspected disclosure of product designs may not permit you to examine every item of personal information on the device.

In this example, you need to determine whether the employee accessed the designs, photographed them, transferred them through a messaging or file-sharing application, uploaded them to cloud storage, or communicated them to an external party. Clearly defining these questions will help you determine which evidence sources and acquisition methods are relevant.

Identification and preservation

Before extensively handling the device, you should document its condition and current state. Record the manufacturer, model, serial number, storage capacity, operating-system version, physical condition, displayed date and time, battery level, power state, lock state, network status, and any connected accessories. SIMs, removable storage, cables, chargers, paired computers, and nearby storage devices may also be relevant when they fall within the authorized scope.

You should pay particular attention to the device's power, lock, and encryption states because they may directly affect the evidence you can acquire. Powering down or restarting the device may make encryption keys unavailable. Leaving it powered on and connected to a network may permit synchronization, incoming communications, or remote commands. You should therefore photograph and record the observed state before deciding how to preserve and isolate the device.

If latent fingerprints, DNA, trace material, or other physical evidence may be relevant, you should coordinate with the appropriate specialists before extensively handling the device.

Preparation

You should develop an acquisition strategy based on the device information and your investigative objectives. Your research should include the exact device model, operating-system version and build, security patch level, encryption configuration, lock state, boot state, storage configuration, pairing status, and available credentials.

In this example, you might determine that the smartphone uses file-based encryption, a locked bootloader, and hardware-backed key protection. These conditions may limit the available acquisition methods and affect whether powering down, restarting, unlocking, or otherwise interacting with the device would reduce access to evidence.

You should compare the available acquisition techniques by considering the data each method can recover, the changes it may introduce, its support for the specific device, and its likelihood of success. You should also verify your tool versions, test the required hardware and cables, confirm that sufficient storage is available, and document the planned procedure before beginning the acquisition.

Isolation

A powered-on mobile device may continue to receive messages, synchronize application data, connect to cloud services, download updates, or process remote-lock and remote-wipe commands. You should isolate the device to reduce these risks while preserving the device state required for acquisition.

No single isolation method is appropriate for every device. Enabling airplane mode requires you to interact with the device and may not disable every wireless interface. Powering it down may make encryption keys unavailable. A shielded container may not block every signal and may increase battery consumption while the device searches for a network.

You should therefore follow a validated procedure based on the device's power state, lock state, available credentials, battery level, signal environment, and planned acquisition method. Minimize and document every interaction with the device. If you use a shielded container or enclosure, verify its effectiveness rather than assuming that the device is completely isolated.

In this example, you should record the device state, select the least disruptive validated isolation method, monitor the battery level, and preserve the state required for your planned acquisition.

Acquisition

During acquisition, you collect data from the device or associated sources using one or more supported methods. Depending on the device and case requirements, you may perform a logical acquisition, backup-based acquisition, filesystem acquisition, physical acquisition, cloud acquisition, or a combination of methods.

Your objective is not simply to obtain the greatest possible volume of data. You should acquire information that is relevant to the investigative questions, within the scope of your legal authority, and through a method whose effects and limitations you can explain.

In this example, you might begin with a supported logical or backup-based acquisition. If it does not provide the required application or filesystem artifacts, you may consider a deeper method supported by the device. You may also examine an authorized cloud account or a paired company computer for backups, synchronization records, or authentication material.

Throughout the acquisition, you should record the tool and version, settings, start and completion times, device state, errors, generated logs, resulting files, and any observed changes to the device. If you use multiple tools or methods, preserve and describe each acquisition separately.

Verification

After acquisition, you should determine whether the process completed successfully and whether the resulting dataset contains the expected files, databases, logs, and artifact categories. Document any errors, unsupported items, inaccessible data, or incomplete sources.

You should generate cryptographic hashes, such as SHA-256, for acquired files or forensic containers where supported. Hashes can demonstrate that a preserved acquisition and its working copies remain unchanged. However, a matching hash does not prove that the acquisition captured every item available on the device.

You should also validate significant artifacts by tracing them to the underlying source data or comparing them across tools where practical. Investigate and document any discrepancies before relying on the affected artifacts in your analysis.

Examination and analysis

During examination, you identify potentially relevant data within the acquisition. During analysis, you interpret that data in relation to the investigative questions. In this example, relevant sources may include messages, emails, photographs, file metadata, browser activity, cloud-storage applications, application logs, location-related records, and deleted or residual artifacts.

You should correlate evidence across multiple sources rather than interpreting individual records in isolation. For example, you might compare a photograph of a product design with its metadata, application-sharing records, message attachments, cloud-upload activity, and communications with external contacts.

You should treat automated tool output as an interpretation of the underlying evidence. When a finding materially supports a conclusion, you should validate it against the original database, file, metadata, acquisition log, or a second independently tested method where practical. You should also distinguish observed facts from your inferences and avoid attributing device activity to a particular person without supporting evidence.

Documentation and reporting

You should maintain documentation throughout the examination rather than creating it only after the analysis is complete. Your examination notes should record device identifiers, observed states, legal authority, handling decisions, acquisition methods, tool versions, settings, timestamps, errors, hash values, validation steps, and any deviations from the planned procedure.

In this example, your final report should explain the investigative questions, the evidence examined, the methods used, and the findings relevant to the suspected disclosure. It should

identify the source of each significant artifact and explain any limitations that affect its interpretation.

You should distinguish among facts directly observed in the evidence, interpretations produced by forensic tools, your own inferences, and your final conclusions. Write the report so that both technical and non-technical readers can understand it, while including enough detail to permit independent review.

Archiving

At the conclusion of the examination, you should preserve the original device, master acquisitions, verified working copies, tool logs, examination notes, reports, screenshots, chain-of-custody records, and cryptographic hashes in accordance with the applicable retention and evidence-management policies.

You should control and document access to the archived material. Preserve enough information to allow the examination to be reviewed, repeated, or extended if new questions arise, additional legal authority is obtained, or improved forensic techniques become available.

This completes the general mobile forensic workflow. The next section examines how differences between Android and Apple mobile platforms affect acquisition and analysis.

Mobile platforms and forensic considerations

The operating system is one of the biggest factors shaping how data is accessed and examined. Today's mobile forensics primarily targets Android and iOS devices:

- **Android**: A Linux-based, open-source OS developed by Google, widely used by various manufacturers. Its openness has led to a vast ecosystem of customizable devices and over 3 million apps on Google Play. Android often allows deeper system-level access, but also comes with fragmented hardware and OS versions, and complex security features like Security-Enhanced Linux (SELinux) and Verified Boot which protect the system integrity. Android's app sandboxing mechanisms isolate applications from each other, but you can often bypass these on rooted devices.
- **iOS**: Apple's Unix-based OS powering iPhones, iPads, and iPods, tightly integrated with hardware and secured by robust encryption and hardware-backed security such as the Secure Enclave. Unlike Android, iOS is a closed system with strict app controls via the App Store. Its data protection classes offer varying levels of encryption for files based on the device's lock state, complicating data access. The secure boot chain ensures only trusted firmware can be loaded.

A thorough understanding of mobile operating systems helps you identify potential sources of evidence, understand security restrictions, and select appropriate acquisition strategies.

However, understanding the operating system is only one part of the forensic process. You must also determine how data will be acquired from the device. Different acquisition methods provide varying levels of access to data, require different levels of technical expertise, and carry different levels of risk. To better understand these approaches, it is useful to examine the forensic pyramid classification system.

The forensic pyramid classification system

Mobile phone forensic acquisition and analysis involve manual effort and the use of automated tools. There are a variety of tools and techniques that are available for performing mobile forensics. All of them have their pros and cons, and it is fundamental that you understand that no single tool is sufficient for all purposes.

When identifying the appropriate method for the forensic acquisition and analysis of mobile phones, a mobile device forensic tool classification system developed by Sam Brothers (shown in *Figure 1.2*) comes in handy:

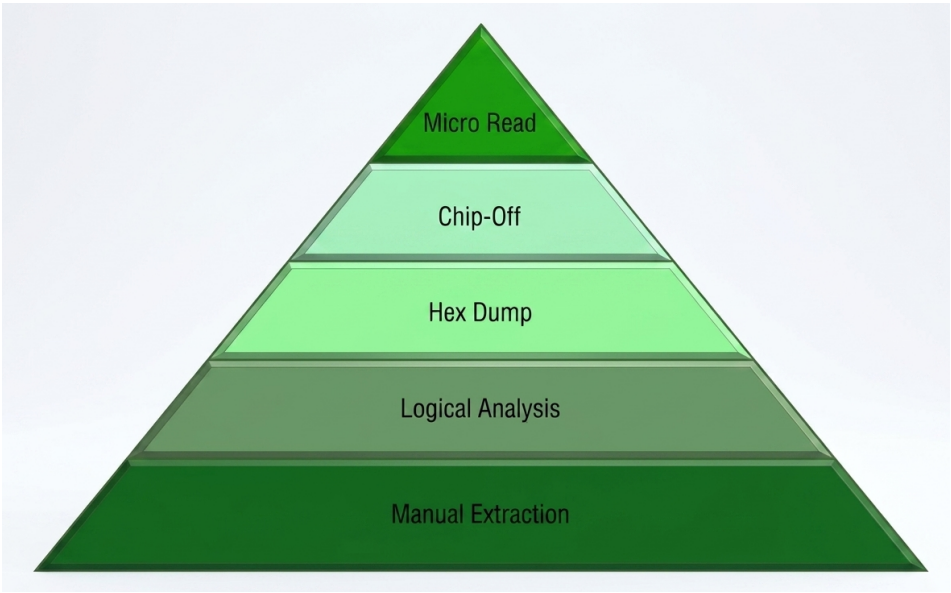


Figure 1.2: Forensic tool classification system

The objective of the mobile device forensic classification system is to enable you to categorize forensic tools and approaches based on the examination methodology of the tool. As you move from the bottom of the pyramid to the top, the methods and tools used for analysis generally become more technical and sophisticated and require longer analysis times.

There are advantages and disadvantages of different techniques used at each layer. As the forensic examiner, you should be aware of these issues before applying a particular technique.

Evidence can be destroyed completely if the given method or tool is not properly utilized. Higher levels of the classification generally require greater technical expertise and may involve more invasive interaction with device hardware or software. However, you should assess risk for the specific device and method rather than infer it solely from the method's position in the pyramid.

Each existing mobile forensic tool can be classified under one or more of the five levels. Let's now discuss each of these approaches.

Manual extraction

The **manual extraction** method involves simply scrolling through the data on the device and viewing the data on the phone directly using the device's keypad or touchscreen. The information discovered is then photographically documented. The extraction process is fast and easy to use, and it will work on almost every phone.

This method is prone to human error, such as missing certain data due to unfamiliarity with the interface. At this level, it is not possible to recover deleted information and grab all the data. There are some tools, such as Project-A-Phone, that have been developed to help an examiner easily document manual extraction. However, this might also result in modification of data, like viewing an unread SMS will mark it as read.

Logical extraction

Logical analysis involves connecting the mobile device to forensic hardware or to a forensic workstation via a USB cable, an RJ-45 cable, infrared, or Bluetooth. Once connected, the computer initiates a command and sends it to the device, which is then interpreted by the device's processor. Next, the requested data is received from the device's memory and sent back to the forensic workstation. Later, you can review the data.

Most of the forensic tools currently available work at this level of the classification system. The **extraction** process is fast and easy to use and requires little training for you. On the flip side, the process may write data to the mobile and might change the integrity of the evidence. In addition, deleted data is not generally accessible with this procedure.

Hex dump / physical extraction

The terms hex dump and physical extraction are closely related and are sometimes used interchangeably. **Physical extraction** refers broadly to acquiring data at the lowest accessible storage level, traditionally as a bit-for-bit copy of the device's storage. A **hex dump** refers more specifically to the resulting raw binary data represented or examined as hexadecimal values and can therefore be considered an output of physical extraction rather than a distinct acquisition method.

Traditionally, physical extraction involved using a custom bootloader, recovery environment, forensic agent, or exploit to access storage at a lower level than logical acquisition. Such acquisitions could contain filesystem structures, application data, unallocated space, and, in some cases, deleted artifacts unavailable through logical extraction.

Modern mobile security has significantly reduced the effectiveness of traditional physical extraction. Mechanisms such as Verified Boot, Secure Boot, hardware-backed encryption, the Secure Enclave, TrustZone, and file-based encryption (FBE) can prevent access to decrypted user data even when raw storage is successfully acquired. Therefore, obtaining a physical image does not necessarily provide usable evidence.

The forensic value of physical acquisition depends on the device state, encryption status, operating system version, hardware security implementation, and available device-specific techniques. Nevertheless, it remains an important acquisition concept because it represents one of the deepest levels of software-based access available to investigators.

Chip-off/ISP

Chip-off and **in-system programming** (**ISP**) are advanced hardware acquisition techniques used when conventional software-based extraction methods are unavailable or unsuccessful. Chip-off involves physically removing the storage chip from the device and reading it using specialized hardware. This technique requires significant expertise because improper handling can permanently damage the chip and destroy evidence. Since the storage device is physically removed, chip-off is generally considered a destructive acquisition method and is typically reserved for severely damaged devices or situations where no alternative acquisition method is available.

ISP provides a less invasive alternative. Rather than removing the storage chip, you connect directly to storage test points on the device's motherboard and acquire data while the chip remains installed. This approach is commonly used with supported eMMC and certain UFS-based devices and may reduce the risk associated with chip removal.

Historically, **joint test action group** (**JTAG**) acquisition was widely used to access device memory through processor debugging interfaces known as **test access ports** (**TAPs**). However, modern device architectures, secure hardware implementations, and the widespread adoption of hardware-backed encryption have significantly reduced the practical applicability of JTAG acquisition. While JTAG remains an important technique from a historical and educational perspective, ISP has largely replaced it in many contemporary hardware acquisition scenarios.

Although chip-off and ISP may provide access to raw storage contents, modern Android and iOS devices typically employ hardware-backed encryption and secure key management systems. Consequently, obtaining a raw memory dump does not necessarily provide access to decrypted user data. For this reason, these techniques are now primarily used in specialized

investigations involving damaged devices, legacy devices, unsupported devices, or cases where other acquisition methods are not feasible.

Micro read

The **micro read** process involves manually viewing and interpreting data seen on the memory chip. The examiner uses an electron microscope and analyzes the physical gates on the chip and then translates the gate status to 0s and 1s to determine the resulting ASCII characters. The whole process is time-consuming and costly, and it requires extensive knowledge and training on memory and the filesystem.

Due to the extreme technicalities involved in micro read, it is only attempted for high-profile cases equivalent to a national security crisis after all other levels of extraction techniques have been exhausted. The process is rarely performed and is not well documented now. Also, there are currently no commercial tools available to perform micro reading.

We will explore these techniques in detail in the later chapters of the book. Each method has implications for the volume and integrity of the recovered data. Forensic tools must also be rigorously tested and validated (for instance, following NIST guidelines for tool validation) to ensure findings stand in court.

Types of evidence extracted from mobile devices

Mobile phones are rich sources of evidence, with data residing in various locations such as the SIM card, external memory cards, and the phone's internal memory. While service providers also retain communication records, forensic investigations primarily focus on data extracted directly from the device's memory.

Although the exact data recovered depends on the phone model and OS, most mobile devices typically store the following categories of information, often tagged with timestamps, which are valuable to investigations:

- **Contacts**: Names, phone numbers, emails, addresses, social media handles, and other address book entries.
- **Call logs**: Records of dialed, received, and missed calls, along with call durations and associated contacts.
- **SMS and MMS**: Text messages plus multimedia content (photos, videos, audio) exchanged via messaging applications.
- **Emails**: Sent, received, and draft messages stored on the device, often from various email clients.
- **Browser history**: Websites visited, search queries performed, cached web data, cookies, and bookmarks.

- **Photos and videos**: Images and videos captured by the phone, downloaded, received, or transferred from other devices. This includes screenshots and screen recordings.
- **Audio files**: Music, voice recordings, and audio messages obtained through downloads or transfers.
- **Documents**: Files created, downloaded, or received through office apps, cloud storage apps, and messaging.
- **Calendar entries**: Scheduled events, appointments, and associated notes.
- **Location data**: Precise GPS coordinates, Wi-Fi access point history, cell tower logs, navigation searches, and map caches showing travel history and frequented locations. This is often a critical artifact.
- **App data**: Extensive data from third-party applications, including the following:
    - **Messaging apps**: WhatsApp, Signal, Telegram, Facebook Messenger, WeChat (chats, media, call logs specific to the app)
    - **Social networking apps**: Facebook, Twitter, Instagram, LinkedIn (posts, direct messages, contacts, media)
    - **Cloud storage apps**: Google Drive, Dropbox, OneDrive (cached files, sync logs)
    - **Financial/payment apps**: Banking apps, Google Pay, Apple Pay, PayPal, cryptocurrency wallets (transaction history, account details, usage logs)
    - **Health and fitness apps**: Data from health trackers, step counts, heart rate, sleep patterns, and medical records
    - **Productivity apps**: Office suites, note-taking apps
    - **Gaming apps**: Scores, in-app purchases, chat logs
- **System logs and event files**: These can provide crucial metadata about device activity, app usage, network connections, battery statistics, and crash reports, often revealing abnormal behavior or malware presence
- **Network connection logs**: Records of Wi-Fi access points connected to, Bluetooth pairings, and VPN connection history
- **Deleted data**: This can include deleted messages or call history etc.

Once the data is extracted from the device, as the forensic investigator, you need to understand how to examine and analyze that data. The next section covers this part.

Examining and analyzing extracted data

This is the crucial phase of the investigation where data extracted from a device is thoroughly examined to uncover evidence relevant to the case. Examination involves applying established

forensic techniques to identify, validate, and isolate meaningful data from large volumes of information that may be present on the device.

Analysis begins with an acquired dataset, which may be a logical extraction, backup, filesystem acquisition, cloud-derived collection, or physical image. Forensic tools parse known structures such as SQLite databases, property lists, media metadata, and application files. File carving may also be attempted when the acquisition contains suitable raw or residual storage data, although encryption and modern flash-storage behavior often limit its effectiveness.

However, effective analysis also requires understanding the context of the case. A fraud investigation may prioritize financial records and communications, while an insider-threat investigation may focus on file access, cloud-storage activity, messaging, and event timelines. The goal is often to perform timeline analysis, reconstructing the sequence of events across various data sources, and link analysis, identifying relationships between individuals, devices, and activities.

A solid grasp of forensic tools is essential. Knowing their features and limitations speeds up analysis and helps you catch issues such as a tool failing to parse certain data because of bugs or unsupported formats. In such situations, expertise in switching tools, manually inspecting data using hex editors, or understanding raw file structures becomes critical. Additionally, since suspects might tamper with, hide, or delete data (anti-forensics), you must be vigilant and know how to handle such scenarios to recover or reconstruct evidence. This includes recognizing and handling mobile malware, which can interfere with data or exfiltrate evidence.

While automated tools generate reports, a human expert's critical review is indispensable. The ability to correlate information from different apps, reconstruct fragmented data, and validate findings against multiple sources distinguishes a thorough analysis.

Legal rules and handling of evidence

As courts increasingly rely on digital data from mobile devices, you must ensure that evidence meets established legal standards. While laws vary by country, the following five core rules generally apply to digital evidence and failing to follow them can render evidence inadmissible:

- **Admissible**: Evidence must be collected in a lawful manner and preserved according to established procedures. This usually means acquiring proper legal authorization, such as a search warrant that specifically covers the seizure and analysis of a mobile device and its data. Evidence obtained without the necessary legal authority, or through improper handling (such as searching beyond what the warrant allows, or exceeding the scope of consent), is often ruled inadmissible. This can cause the entire investigation to collapse.

> **Tip**
>
> Always document the legal basis for acquiring and analyzing a device, whether it is a warrant, consent, or an exception such as exigent circumstances.

- **Authentic**: Evidence must clearly be tied to the incident under investigation. This means proving that the data came from the specific device seized and that it was not tampered with or mixed with unrelated information. Maintaining a solid chain of custody is essential. Each handoff of the device or evidence must be documented, including who handled it, when, and why. Cryptographic hashes are crucial here.

> **Tip**
>
> Keep detailed logs and photographs of the device upon receipt, noting serial numbers, appearance, and any distinctive features. This supports authenticity if questioned later.

- **Complete**: Evidence must present the full picture, not just selective parts that support one argument. Incomplete or cherry-picked data can mislead, potentially resulting in wrongful conclusions or justice being denied. A thorough forensic examination looks at all potentially relevant data, even if some findings do not support the case theory (including exculpatory evidence).

> **Tip**
>
> Include exculpatory data (data that may clear a suspect) in your reports. Omitting such data could be seen as misconduct and damage credibility.

- **Reliable**: The methods and tools used to collect and analyze evidence must be dependable and scientifically accepted. Processes should be repeatable by other experts, yielding the same results. Using untested or flawed tools can lead to evidence being challenged. For example, if a forensic tool is known to corrupt data or miss certain artifacts, its findings may be questioned. This is especially critical for advanced or destructive methods like chip-off or JTAG. These should be documented carefully, and only performed by trained professionals, because improper execution can destroy evidence. Tool validation frameworks like NIST mobile device forensics guidelines are crucial.

> **Tip**
>
> Regularly validate forensic tools against known test data sets to prove they work correctly and keep records of these validations.

- **Believable**: You must be able to clearly explain what was done, why it was done, and how the integrity of the evidence was protected. Findings should be presented in a logical, easy-to-understand manner that makes sense to judges and juries, who often lack technical expertise. The forensic examiner often serves as an expert witness.

> **Tip**
>
> Prepare for testimony by practicing how to explain technical methods in simple language, avoiding jargon. Make sure you can walk through your documentation and processes step by step.

Now, let's look at best practices to follow during these investigations.

Best practices for mobile forensic investigations

Good forensic practices are the backbone of any credible digital investigation. Improper handling can compromise evidence and make it inadmissible in court. Even minor mistakes, whether intentional or accidental, can alter critical data and weaken a case. Following rigorous and standardized practices is essential for all forensic examiners.

Securing evidence

Modern smartphones include features such as Find My iPhone, Android's Find My Device, and remote wipe capabilities. These allow owners or attackers to erase data remotely. Additionally, as long as a device is powered on and connected, it continues to receive new data that could overwrite or change existing information.

Follow these steps to protect the evidence:

1. Isolate the device immediately using Faraday bags or shielding enclosures that block network signals. If possible and if the device is unlocked, switch it to airplane mode. Avoid taking actions that could alter data if the device is locked.
2. If the phone is connected to a computer for syncing or charging, do not disconnect it abruptly. First, capture a memory image of the computer, which may contain logs or temporary files linked to the device.

3. Collect all peripherals, memory cards including SD cards and SIM cards, cables, chargers, docks, and nearby computers or drives. These may contain backups or other related data.
4. In some cases, collecting fingerprints or other biometrics is necessary to link the device to a person, often performed before any other handling

Properly securing a device is the first step in protecting potential evidence from loss, modification, or remote access. Once the device has been secured, the next priority is ensuring that the evidence remains unchanged throughout the examination process.

Preserving evidence

Evidence must be kept in its original state to be accepted by a court. Forensic guidelines recommend following these practices:

- Avoid working directly on original evidence. Instead, create a forensic image (bit-by-bit copy) of the device's storage.
- Use cryptographic hashes to verify the integrity of the data. Compute hashes like SHA-256 or SHA-512 for both the original acquired image and the working copies. This demonstrates they are identical. Re-verify these hashes at different stages of the investigation.
- Store the original device and the master copies in sealed, secure storage with restricted access. Perform all examinations on verified copies using a dedicated, forensically sound workstation that is isolated from external networks.
- Any change to evidence, no matter how minor, must be documented and explained.

Preserving evidence helps maintain its integrity and admissibility throughout the investigation. Equally important is documenting every action taken, ensuring that the examination process can be reviewed, validated, and reproduced if necessary.

Documenting the evidence and changes

Detailed documentation is critical to ensure your work can be reviewed or repeated by another examiner. This strengthens the credibility of your findings. Follow these practices to ensure the same:

- Take photographs of the device before touching it, including its surroundings and any connected hardware. Continue documenting the device from seizure to transport to lab intake.
- Keep careful, contemporaneous notes of every action taken. Record who handled the device, when, what tools were used (including version numbers), what settings were applied, and observations such as whether the phone was on or locked.

- Note any changes made. For example, if the forensic tool used breaks a disk image into segments or if the device had to be powered off and on, this must be logged.
- Maintain a chain of custody log that proves the evidence remained secure and unaltered from collection to presentation in court.

Thorough documentation provides a complete record of the investigation and supports the credibility of the findings. The final step is to communicate those findings clearly through a well-structured forensic report.

Reporting

A strong forensic report describes what was done, how it was done, and what was found. It should include evidence that both supports and challenges the case narrative (including exculpatory findings). A clear, balanced report is more persuasive than one that only presents selective findings.

Typically, a forensic report includes the following details:

- Case identifiers such as agency name, case number, investigator, and submitter
- Date the evidence was received and a detailed list of items with serial numbers, make, model, and physical condition
- Information on the tools and software used, including version numbers and validation records
- A step-by-step record of acquisition and examination procedures
- Chain of custody records
- Findings from the investigation, such as call logs, SMS, app data, deleted files, location history, and recovered artifacts, are often presented with timeline analysis and link analysis
- Photos or screenshots captured during the process
- Hash values to show data integrity
- A clear conclusion explaining how the findings relate to the investigation

Together, these practices help ensure that mobile forensic examinations remain technically sound, legally defensible, and transparent. Following them consistently improves the reliability of the investigation and the evidentiary value of the findings.

Most forensic tools can generate automatic reports, but these should always be reviewed, edited, and supplemented with your professional notes, as well as an explanation of any manual analysis performed.

Effective forensic examiners work with care, maintain transparency, and prepare for close legal scrutiny. They not only know how to extract and analyze data but also ensure that every step

can be defended as legally and scientifically sound. By following these best practices, you protect the evidence, support the investigation, and help ensure your work stands up in court. Following established forensic procedures and best practices significantly improves the likelihood of a successful investigation.

However, mobile forensic examinations remain inherently challenging due to the rapid evolution of mobile technology, security architectures, and user behavior. Modern devices incorporate sophisticated security controls, diverse hardware platforms, cloud integrations, and encryption mechanisms that can complicate data acquisition and analysis. As a result, even experienced investigators may encounter obstacles that limit access to evidence or increase the complexity of an examination.

The following section discusses some of the most significant challenges faced during modern mobile forensic investigations.

Challenges in modern mobile investigations

Mobile forensics introduces a distinct set of challenges that make it more complex than traditional computer forensics. One of the most significant hurdles is that data on mobile devices is highly volatile, that is, it can be easily altered, deleted remotely, or spread across multiple synchronized platforms. As mobile devices increasingly integrate cloud services, IoT systems, and encrypted communication apps, preserving evidence requires more diligence than ever before. Law enforcement and forensic investigators frequently encounter obstacles that complicate acquiring and analyzing mobile data.

Let's look at some of the key challenges in modern mobile investigations:

- **Hardware diversity**: The mobile device market is saturated with countless models from a vast array of manufacturers. Devices vary widely in hardware configurations, chipsets (such as, Qualcomm, MediaTek, and Apple A-series), storage technologies (including eMMC, UFS, and NVMe), and peripheral features. New models are introduced on a near-monthly basis, each with subtle or substantial differences. This short product lifecycle means you must continuously adapt and update their knowledge, tools, and accessories to keep up.
- **Operating system fragmentation**: Unlike personal computers where Windows long dominated, mobile devices run on a mix of operating systems. While Android and iOS now control most of the market, legacy platforms like BlackBerry OS, Windows Phone, and niche embedded systems still surface in investigations. Even within Android, the open-source nature leads to significant fragmentation due to diverse vendor implementations (Samsung's One UI, Xiaomi's MIUI, and so on), often resulting in unique file paths, app data storage, and security tweaks that impact forensic analysis.

For both Android and iOS, frequent OS updates and security patches can change how data is stored, encrypted, or accessed, further complicating forensic workflows and often breaking existing acquisition methods.

- **In-built security features**: Modern mobile platforms are deliberately designed to protect user data through sophisticated security architectures. Features like full-disk encryption (FDE) and particularly FBE, Secure Enclaves/TrustZones, hardware-backed key stores, biometric locks, app sandboxing, and signed firmware/Verified Boot are excellent for user privacy but present major obstacles to forensic extraction. Take these two security features for example:
    - **Secure Enclave** (iOS) or **TrustZone** (Android) are dedicated, isolated hardware subsystems that handle cryptographic operations and protect sensitive data (like encryption keys and biometric information) even if the main processor is compromised. This makes it exceedingly difficult to bypass passcode protection without the user's explicit input or a highly specific hardware vulnerability.
    - **Bootloader** locking and signed boot chains prevent the flashing of unauthorized firmware or custom recovery, which are often used in advanced acquisition techniques.

Note

A defining case that shows how encryption can limit investigations is the Apple vs. FBI dispute after the 2015 San Bernardino shooting. The FBI recovered an iPhone 5C used by one of the attackers but couldn't access its data due to iOS encryption tied to a secure enclave and passcode. The FBI sought a court order forcing Apple to create a special iOS to bypass these protections, but Apple refused, arguing it would set a dangerous precedent by weakening security for all users. The standoff sparked a global debate on privacy vs. security. Eventually, the FBI paid a third party to unlock the phone. This case still shapes discussions on device encryption and lawful access today.

- **Preventing data modification**: A foundational rule in digital forensics is that the original data should remain unaltered. Yet on mobile devices, this is exceptionally difficult to guarantee. Simply powering on a phone can trigger background processes that modify system files or logs. Even phones appearing to be off might run low-level processes; many still allow alarms to sound or system checks to run. Any state change risks unintentionally modifying or deleting evidence.

- **Anti-forensics and user manipulation**: Suspects increasingly are employing anti-forensic techniques to thwart investigations. These include secure deletion apps, data obfuscation like steganography, using hidden or encrypted containers such as VeraCrypt on mobile, app cloning/hiding mechanisms, and employing software that forges or scrambles metadata. Some may even root or jailbreak devices, install stealth apps, use burner phones, or intentionally modify the OS, complicating evidence authenticity and recovery. Ephemeral messaging apps such as Snapchat and Signal with disappearing messages and which are designed to auto-delete content also pose a significant challenge.
- **Passcode and biometric locks**: Accessing a locked device remains one of the most common roadblocks. While there are forensic techniques and tools that can bypass or brute-force certain screen locks, these methods do not work universally especially against newer OS versions and devices with strong hardware-backed encryption. Failures here can completely block access to crucial evidence.
- **Tool and accessory challenges**: Given the vast diversity of devices, no single tool covers all makes and models. You will often need a toolkit of different commercial forensic solutions such as Cellebrite UFED, MSAB XRY, Magnet AXIOM, or Oxygen Forensics Detective, open-source tools like Autopsy, ADB, and various Python scripts, plus an inventory of practical accessories including specialized USB cables, adapters, replacement batteries, and proprietary chargers to interface with various devices. Keeping this inventory current is both costly and logistically challenging.
- **Dynamic, fragile evidence**: Digital evidence on a mobile device can change rapidly, sometimes by simply opening an app. Actions like previewing chats, images, or browser history can update timestamps, alter cache data, or trigger auto-syncs. Additionally, accidental factory resets or OS restores sometimes triggered by inattentive handling can permanently erase data.
- **Communication shielding**: Because mobile devices actively communicate over cellular networks, Wi-Fi, Bluetooth, and even NFC, there's a risk that a seized phone could receive a remote wipe command or sync new data. Placing devices in airplane mode and Faraday bags is essential, yet even then, weak shielding or overlooked radios (like eSIM updates or secure element communications) pose risks.
- **Malware and rogue applications**: Some mobile devices may be infected with malware designed to tamper with data, exfiltrate evidence, or destroy it. These malicious programs can spread across interfaces (wired or wireless), putting not just the evidence at risk, but potentially contaminating forensic workstations if proper isolation isn't

maintained. Mobile malware forensics is a specialized sub-field focusing on identifying, analyzing, and mitigating these threats.

- **Legal and jurisdictional complexities**: Mobile evidence often crosses borders. A suspect might live in one country, store data in cloud services located in another, and communicate with people in entirely different regions. This creates multi-jurisdictional challenges. You must carefully navigate regional laws, privacy regulations such as the GDPR, and international cooperation agreements like mutual legal assistance treaties (MLATs) to access or use this data in a legally valid way. Cloud data seizure often requires separate legal processes from on-device data.

While these challenges can significantly complicate mobile forensic investigations, they are not insurmountable. Successful examinations require a combination of technical expertise, sound forensic methodology, appropriate tools, and a thorough understanding of modern mobile platforms. As mobile devices continue to evolve, you must continually adapt your techniques and knowledge to ensure that valuable evidence can be acquired, analyzed, and presented in a reliable and legally defensible manner.

Summary

In this chapter, you learned the fundamental concepts of mobile forensics and why mobile devices have become one of the most important sources of digital evidence in modern investigations. We examined how smartphones store a wide range of information, including communications, location data, photographs, application artifacts, and cloud-synchronized content, making them valuable sources of evidence in criminal, civil, corporate, and intelligence investigations.

You learned the key phases of the mobile forensic workflow, from evidence intake and device identification through acquisition, analysis, reporting, and long-term preservation. We also discussed the importance of following forensically sound procedures, maintaining chain of custody, preserving evidence integrity, and ensuring that findings can withstand legal scrutiny.

The chapter introduced the major mobile operating systems encountered during forensic investigations and provided an overview of common acquisition approaches, ranging from manual and logical extraction to advanced hardware-based techniques. We also explored the types of evidence commonly found on mobile devices and examined the role of forensic analysis in transforming raw data into meaningful investigative findings. Finally, we discussed best practices for conducting mobile forensic investigations and examined many of the challenges faced by modern investigators, including device diversity, encryption, anti-forensic techniques, cloud services, and evolving security protections.

With these foundational concepts in place, the following chapters will build upon this knowledge by exploring platform-specific forensic techniques, acquisition methods, data analysis, and evidence recovery in greater detail.

References

1. Ericson Mobility report: `https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/mobile-traffic-forecast`
2. 2022 Idaho murders: `https://en.wikipedia.org/wiki/2022_Idaho_student_murders`
3. Apple vs FBI dispute: `https://en.wikipedia.org/wiki/Apple%E2%80%93FBI_encryption_dispute`

2

iOS Architecture, Security, and Filesystem Overview

As of 2026, Apple reports an installed base of more than 2.5 billion active devices worldwide. More than 1.5 billion of these are active iPhones running iOS, making Apple's mobile ecosystem one of the largest and most significant sources of digital evidence for forensic investigations. While Android continues to dominate global smartphone market share, iOS devices are a frequent and critical component in forensic investigations.

To perform a forensic examination effectively, it is essential to understand the internal architecture and operational model of iOS devices. This includes knowing how the operating system functions, how data is stored and encrypted, and what technical and security controls affect data access. A strong grasp of these elements directly influences what data can be acquired, how it should be handled, and which acquisition techniques are most appropriate.

This chapter serves as your foundation for iOS forensics. Although the core architecture of iOS has remained consistent over the years, this chapter reflects the security model and platform enhancements present in modern releases, including iOS 26. Before diving into acquisition or analysis, we will walk through key concepts such as iOS architecture, the secure boot process, data protection classes, file system layout, and how modern security enhancements impact forensic workflows. These fundamentals are critical not only for gaining access to evidence but also for ensuring your methods are held up in a legal context.

In this chapter, you will learn about:

- Introduction to the iOS ecosystem
- iOS architecture layers
- iOS security model

- Keychain and iCloud security
- Apple file system (APFS) overview
- Impact of modern security enhancements on forensics
- Jailbreaking and its forensic implications

Throughout this book, we will cover forensic acquisition and analysis techniques across various iOS devices. While the iPhone remains the primary focus due to its prevalence, many principles also apply to iPads and Apple Watches. This foundational knowledge will equip you to approach iOS investigations with confidence and technical precision.

Introduction to iOS ecosystem

Apple's **iOS ecosystem** is a tightly integrated and highly controlled environment that includes both hardware and software. Unlike many other mobile platforms, Apple designs its own hardware (SoCs, Secure Enclaves, and custom silicon), builds the operating system (iOS, iPadOS, and watchOS), and controls the distribution of applications through the App Store. This vertical integration gives Apple complete control over security, user experience, and data storage, which significantly influences digital forensic investigations.

At the center of the mobile ecosystem is iOS, which shares foundational technologies with macOS. Both are built on Darwin, an open-source operating-system foundation that combines the XNU kernel with components derived from Mach and BSD.

For forensic analysts, Apple's approach presents both opportunities and challenges. On the one hand, the uniformity of Apple's hardware and software simplifies tool development and testing, especially when compared to the fragmented Android ecosystem. On the other hand, Apple's strict controls, proprietary connectors, encrypted storage, and secure boot processes often make forensic access more difficult. In many cases, physical access to data without a passcode is extremely limited. Apple's integrated security architecture can significantly restrict access to protected data, particularly when the passcode is unavailable. As discussed in *Chapter 1*, this tension between device security and lawful access has played a prominent role in public and legal debates.

Key elements of the iOS ecosystem that you as forensic examiners must understand include:

- **iOS versions and supported devices**: Each version of iOS supports specific hardware models. Older devices may be vulnerable to publicly known exploits (such as bootrom vulnerabilities affecting certain chipsets), whereas newer devices incorporate stronger hardware-backed security features that significantly limit forensic acquisition options. As a result, tool support often depends on both the device model and the installed iOS version. Conversely, devices running the latest versions of iOS, including iOS 26,

generally provide the fewest publicly available acquisition options because of continually evolving hardware and software protection.

- **Apple ID and iCloud**: The Apple ID is central to the user's identity and ties together backups, messages, photos, and app data across devices using iCloud. Gaining lawful access to iCloud data can be an important source of evidence when on-device acquisition is not possible. Users may also enable **advanced data protection (ADP)**, which extends end-to-end encryption to many additional categories of iCloud data. When ADP is enabled, Apple cannot decrypt or provide this protected data, which can significantly limit the evidence available through lawful requests. Understanding whether ADP is enabled is therefore an important consideration during an iOS forensic investigation.

> **Tip**
>
> Never evaluate acquisition options based on the iOS version alone. Record the exact device model, chip generation, iOS version and build, lock state, and available authentication material before selecting an acquisition method. Two devices running the same iOS version may support very different acquisition techniques.

- **App Store and sandboxing**: Each application normally operates within its own container and has limited access to data belonging to other applications. Authorized data sharing may occur through system APIs, entitlements, App Groups, extensions, user-selected files, or other controlled mechanisms. During analysis, you may therefore need to examine an application's primary container, shared containers, preferences, caches, and related system records.
- **Secure Enclave**: Supported Apple devices use the Secure Enclave as an isolated security subsystem for sensitive operations involving passcodes, biometrics, and cryptographic key management. Its forensic significance is discussed later in this chapter.
- **Secure Boot chain**: Every iPhone verifies each stage of the boot process using cryptographic signatures before executing code. This chain of trust helps prevent unauthorized firmware or operating systems from loading and plays a significant role in limiting low-level forensic acquisition techniques.

A clear understanding of how the iOS ecosystem functions is critical for effective forensic work. The choice of tools, acquisition methods, and the level of access you can achieve will depend on various factors such as the device model, iOS version, security configuration, and whether

iCloud is enabled. Now that we've explored the key components of the iOS ecosystem, let's look at the underlying architecture that makes these features possible.

iOS architecture layers

iOS is built on a layered architecture that abstracts the complexity of hardware interaction and provides a controlled, secure, and scalable environment for apps. This layered model enables Apple to manage system performance and security while offering developers APIs that promote uniform behavior across devices.

Understanding the different layers of the iOS architecture is crucial for forensic analysts, as each layer plays a role in how data is stored, accessed, protected, or sandboxed. These layers also help pinpoint where various artifacts may reside or how they might be obfuscated. Apple categorizes the iOS architecture into four primary layers, each serving a distinct role in the operation of the operating system:

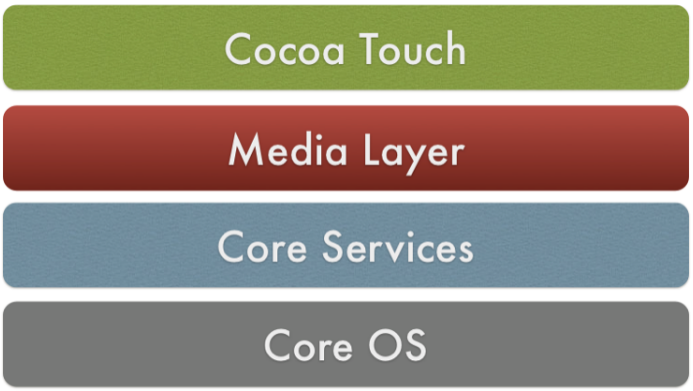


Figure 2.1: IOS high-level architecture overview

Each layer of the iOS architecture has a distinct role in the operation of the operating system and presents unique forensic considerations. In the following sections, we will examine each layer in detail, exploring its primary components, responsibilities, and relevance to forensic investigations.

Core OS layer

Core OS layer is the lowest level and closest to the hardware. It includes the XNU kernel (a hybrid kernel derived from Mach microkernel and components of BSD), device drivers, filesystems, and the foundational security infrastructure, including sandboxing, code signing, and encryption. One of the most critical components of this layer is the SEP, which handles

sensitive operations such as passcode management, biometric authentication (touch ID/face ID), and key generation/storage. The SEP operates independently of the main processor, running its own microkernel-based OS.

From a forensic perspective, the core OS layer defines many of the boundaries that determine what can be accessed during acquisition. Sensitive operations involving passcode verification, biometric authentication, and cryptographic key management are protected by the Secure Enclave and hardware-backed security mechanisms. Accessing data protected by these mechanisms generally requires the appropriate authentication state, cryptographic keys, or a device-specific acquisition technique.

Core services layer

Core services layer provides essential services such as networking, threading, file access, and inter-process communication. It also includes key frameworks like CoreFoundation, CFNetwork, and SQLite. Applications rely on this layer for storing data, accessing preferences, and interacting with local or remote resources.

Forensic tools often extract artifacts from this layer, such as SQLite databases (like SMS, call history, Safari history), Property List (plist) files and Preference files stored in the sandboxed app containers (such as in Library/Preferences).

Media layer

Media layer provides audio, video, and graphics rendering capabilities. Frameworks like AVFoundation, CoreAudio, and CoreGraphics live here. While it may seem less relevant to forensics, media metadata (EXIF data in photos, timestamps in video files, cached media content) often contain important timeline and location evidence.

Cocoa touch layer

At the top is the **cocoa touch layer**, which contains the UIKit framework for UI development and the frameworks that support gesture recognition, multitasking, and push notifications. Applications heavily interact with this layer, which defines how data is presented and managed by the user. Artifacts generated here include application-specific caches, notification logs, and user preferences, which are useful when attempting to reconstruct user activity.

The layered architecture of iOS provides the foundation for application execution, system services, and hardware interaction. However, architecture alone does not explain why iOS is regarded as one of the most secure mobile operating systems. Apple complements this architecture with multiple layers of hardware and software security designed to protect user data, enforce application isolation, and maintain system integrity. Understanding these security mechanisms is essential for forensic examiners, as they directly influence data acquisition, evidence accessibility, and techniques that can be used during an investigation.

iOS security model

Apple has built a multi-layered security model that integrates hardware and software components tightly within the iOS ecosystem. For forensic investigators, understanding the iOS security model is critical for identifying what types of data can be recovered, under what conditions, and through what techniques.

This section focuses on three major components of the iOS security model that are relevant to forensic investigations: **hardware security**, **system security**, and **application security**.

Hardware security

At the heart of iOS security lies hardware-based encryption, which ensures that sensitive data is protected even if the device falls into the wrong hands. Two key hardware elements include the Secure Enclave and UID/GID encryption keys.

Secure Enclave processor (SEP)

Secure Enclave is an isolated security subsystem integrated into supported Apple systems on a chip. It runs its own protected operating environment and participates in sensitive operations involving passcode processing, biometric authentication, data protection, and cryptographic key management. Its exact capabilities vary by hardware generation.

The Secure Enclave helps enforce passcode-attempt delays and protects key material from direct access by the main operating system. If the user has enabled the **Erase Data** option, iOS can render passcode-protected data inaccessible after ten consecutive failed passcode attempts. These protections make straightforward offline passcode guessing impractical and significantly constrain forensic acquisition.

UID and GID keys

Each Apple system on a chip contains hardware-bound cryptographic material, including a device-unique **UID key**. Apple also uses **GID keys** associated with a class of processors. These keys are not directly available to software and participate in hardware cryptographic operations and key derivation.

The UID helps bind protected data to the specific device. As a result, copying encrypted storage to another device does not provide the hardware-bound material required to decrypt it. The UID should therefore be described as a hardware-bound key used by cryptographic architecture, not as a key that a forensic tool can extract or transfer.

Memory integrity enforcement

Apple introduced **memory integrity enforcement** (**MIE**) on supported newer hardware alongside iOS 26. MIE combines hardware and operating-system protections intended to make

memory-corruption exploitation more difficult. It complements existing mechanisms such as Pointer Authentication Codes, Kernel Text Read-Only Region protections, and execution controls, although the available protections vary by device generation.

From a forensic perspective, stronger memory-integrity protections may reduce the reliability or availability of exploit-based acquisition methods on supported devices. Their effect depends on the exact hardware, iOS build, vulnerability, and acquisition technique; MIE should not be described as eliminating kernel exploitation altogether.

System security

Apple's system security covers critical processes such as the Secure Boot process, software update integrity, and the protection of core components like the CPU, memory, storage, applications, and user data.

Secure Boot chain

At the heart of iOS system security is the **Secure Boot chain**, a process that verifies the integrity and authenticity of each software component as the device starts up. It begins with immutable code embedded in the hardware called the Boot ROM, which is trusted by default. The Boot ROM establishes the hardware root of trust and verifies the next stage of the boot process. Each subsequent stage verifies the cryptographic signature and integrity of the software that follows it, ultimately establishing a chain of trust through the bootloader, kernel, and operating system components.

At every step, cryptographic signatures are validated to ensure the code has not been tampered with. If any part of this chain is compromised or unsigned, the boot process halts, preventing untrusted software from running. This chain of trust ensures that only Apple-signed and verified components are loaded at every stage of the boot process, making it extremely difficult for malware or modified operating systems to persist on a device.

> **Note**
>
> As of iOS 26, no publicly documented mechanism exists for bypassing the Secure Boot chain on current-generation Apple devices without relying on a previously unknown vulnerability.

Software updates

To maintain system integrity after boot, iOS also enforces strict rules around **software updates**. Updates can only be installed if they are cryptographically signed by Apple. Devices also include protections to prevent downgrading to older, potentially vulnerable versions of

iOS. This prevents attackers from installing outdated firmware to exploit known vulnerabilities.

Kernel integrity

The **kernel** is the core of the iOS operating system. It manages memory, schedules processes, controls communication with hardware, and enforces many of the operating system's security policies. Because the kernel operates with the highest level of privilege, compromising it could allow an attacker to bypass security controls, access protected data, or execute unauthorized code. To prevent this, Apple has introduced multiple kernel protection mechanisms that work together to preserve system integrity and make kernel-level attacks significantly more difficult:

- **Kernel text read-only region** (**KTRR**): Introduced with devices based on the A10 chip and later, KTRR protects the kernel's executable code by marking it as permanently read-only after the system has booted. Once enabled, even code running with kernel privileges cannot modify protected kernel memory. This significantly increases the difficulty of developing persistent kernel exploits and modern jailbreaks that rely on patching the kernel.
- **Apple mobile file integrity** (**AMFI**): AMFI enforces Apple's code-signing policy by verifying that executable code has been digitally signed by Apple or an authorized developer before it is allowed to run. If an application or system component fails signature verification, iOS prevents it from executing. This mechanism prevents unauthorized software, malware, and modified system binaries from running on the device. Most jailbreak techniques attempt to bypass or disable AMFI to allow unsigned code to execute.
- **Pointer authentication codes** (**PAC**): Modern Apple processors include hardware support for pointer authentication, a security feature that protects important memory pointers from being modified by attackers. The processor generates and verifies cryptographic signatures for selected pointers before they are used. If a pointer has been tampered with, authentication fails and execution is stopped. PAC helps mitigate many memory corruption attacks that previously enabled privilege escalation and kernel exploitation.
- **Data execution prevention** (**DEP**): DEP prevents data stored in memory from being executed as program code. Memory regions intended for storing data, such as the stack and heap, are marked as non-executable. As a result, even if an attacker injects malicious code into these regions, the processor refuses to execute it. DEP forces attackers to rely on more sophisticated exploitation techniques and provides an additional layer of protection against arbitrary code execution.

Application security

Apple complements its hardware and system security with a comprehensive application security model. Every application is isolated from the operating system and from other applications, ensuring that a compromised or malicious app cannot freely access sensitive data elsewhere on the device. This model is enforced through mechanisms such as application sandboxing, mandatory code signing, and entitlements, all of which play an important role in determining what you can and cannot access during a forensic examination.

App sandboxing

Each iOS application runs within its own sandbox, a restricted environment that isolates the application's files, memory, and system resources from the rest of the operating system. By default, an application can only access its own data directory and has no visibility into the data stored by other applications. Access to protected resources such as contacts, photos, camera, microphone, calendars, or location services requires explicit user permission and appropriate application entitlements.

Sandboxing significantly reduces the impact of malware and prevents one compromised application from accessing another application's data. From a forensic perspective, it also means that application data is typically stored within isolated containers. You must therefore locate and analyze each application's sandbox individually when recovering artifacts such as messages, documents, cached files, or configuration data.

Tip

Do not rely only on an application's display name when locating its forensic artifacts. Identify the application by its bundle identifier and examine its data container, shared App Group containers, preferences, caches, and related databases. Evidence associated with a single application may exist in several locations.

Code signing

Apple requires all executable code running on iOS devices to be digitally signed using certificates issued through Apple's **code-signing** infrastructure. Before an application or system component is executed, iOS verifies its digital signature to ensure that it has not been modified and originates from a trusted source. If signature verification fails, the operating system prevents the code from executing.

Recent versions of iOS, including iOS 26, continue to strengthen runtime code integrity checks, making unauthorized code execution increasingly difficult even after an initial vulnerability has been exploited.

Mandatory code signing significantly reduces the risk of malware and unauthorized software running on the device. Most jailbreak techniques rely on exploiting vulnerabilities that bypass or disable code-signing enforcement, allowing unsigned code to execute. For forensic investigators, this restriction also limits the ability to deploy custom forensic utilities directly onto modern iOS devices without exploiting a security vulnerability.

Entitlements

Entitlements are special permissions embedded within an application's code signature that define which privileged system capabilities the application is allowed to use. Examples include access to iCloud services, HealthKit, Apple Push Notification Service (APNs), Keychain sharing, App Groups, and background processing. Certain entitlements are reserved exclusively for Apple system applications, while others require explicit approval from Apple during the application signing process.

From a forensic perspective, entitlements provide valuable insight into an application's capabilities and intended behavior. Examining an application's entitlements can help you understand which system resources the application was authorized to access, how it interacts with other applications or services, and where relevant forensic artifacts are likely to be stored. This information can also assist in identifying suspicious or unusually privileged applications during an investigation.

The security mechanisms discussed so far primarily govern how applications execute and access system resources. The next section shifts the focus to how iOS securely stores and synchronizes sensitive user data through the Keychain and iCloud security architecture.

Keychain and iCloud security

The **iOS Keychain** is a secure credential storage system used to protect sensitive information such as usernames, passwords, cryptographic keys, digital certificates, Wi-Fi credentials, VPN settings, authentication tokens, and other secrets. Although the underlying data is stored in a SQLite database, it is protected using hardware-backed encryption and access controls that tie its availability to both device-specific cryptographic keys and user authentication.

Each Keychain entry consists of a data value (such as a saved password) and metadata that describes its attributes such as the associated service, account identifier, creation timestamp, and accessibility level. Applications store and retrieve items through a controlled API, and they are only permitted to access items that they own or that are shared via specific entitlements.

Keychain availability varies significantly by acquisition method, device state, and the accessibility class assigned to each Keychain item. Some Keychain material may be included in encrypted local backups or recovered through supported forensic acquisition methods, while

more strongly protected items remain dependent on device-specific cryptographic keys and authentication state.

Access groups and access control

One of the most important security features of Keychain is its use of **access groups** and **access control** policies to determine which apps or processes are allowed to access a particular Keychain item. Each Keychain item is associated with an access group, which is essentially a logical grouping of apps that are allowed to share secure data. For example, a set of apps from the same developer or using the same provisioning profile can share a login token using the same access group.

ACLs provide fine-grained control over when and how data can be accessed. This includes restrictions such as requiring biometric authentication (face ID or touch ID), limiting access to only when the device is unlocked, or allowing background access under certain conditions. These access policies are enforced by the operating system and, in some cases, the Secure Enclave itself.

This means that even if a malicious app gains execution on the device, it cannot freely read other apps' Keychain entries unless explicitly authorized. From a forensic standpoint, this segmentation also limits how much information can be extracted unless the app context and permissions are understood and reproduced.

iCloud Keychain and end-to-end encryption

To enhance usability, Apple introduced iCloud Keychain, a feature that synchronizes Keychain data across the user's Apple devices. This allows users to have seamless access to saved passwords, credit card information, and Wi-Fi credentials on all their devices. However, Apple's design ensures that this convenience does not come at the expense of security.

iCloud Keychain data is protected using **end-to-end encryption**, meaning that it is encrypted on the source device, transmitted in encrypted form, and only decrypted on the target device. Importantly, Apple cannot decrypt this data, as the encryption keys are derived from the user's device passcode and a complex chain of trust established through **Secure iCloud Key Escrow**. This system uses iCloud Key Vaults and escrowed key material to authenticate and verify the user's identity without ever exposing the actual Keychain contents to Apple. In addition to passwords, passkeys created using Apple's Passwords app are also synchronized through iCloud Keychain using end-to-end encryption.

To enroll a new device in iCloud Keychain, the user must approve it from an existing trusted device or use two-factor authentication. This secure introduction process prevents impersonation or unauthorized device access. Beginning with recent iOS releases, including iOS 26, users manage many of these credentials through Apple's dedicated Passwords app, which provides a unified interface for passwords, passkeys, verification codes, and shared credentials while continuing to rely on the underlying iCloud Keychain infrastructure.

Users may also enable ADP, discussed earlier in this chapter, to extend end-to-end encryption to additional categories of iCloud data, including iCloud backups, Photos, Notes, and other supported services. When enabled, Apple no longer possesses the encryption keys required to decrypt this data, significantly reducing the amount of cloud-based evidence available through lawful requests. Consequently, determining whether ADP is enabled has become an important step during modern iOS forensic investigations.

Forensic relevance and extraction challenges

From a forensic analysis perspective, the Keychain and iCloud Keychain present both valuable opportunities and significant barriers. On an unlocked device, it is often possible to extract at least some Keychain data using tools such as Cellebrite UFED, Magnet AXIOM, or Elcomsoft iOS Forensic Toolkit. However, data protected by more restrictive access controls such as "When Unlocked" or "After First Unlock" may still be inaccessible depending on the device state.

On the relatively small number of devices that can still be jailbroken, as investigators, you might be able to extract a more complete Keychain database, including authentication tokens, credentials, certificates, and other protected secrets. However, modern Apple silicon devices running current versions of iOS have significantly reduced the availability of public jailbreaks.

iCloud Keychain adds another layer of complexity. Even if you gain access to a user's iCloud backup via credentials or lawful request, the Keychain data is typically excluded or remains encrypted. Access to iCloud Keychain data depends on the available authentication material, trusted devices, account security configuration, and acquisition technique. Possession of iCloud credentials or access to an iCloud backup alone should not be assumed to provide access to synchronized Keychain secrets.

For forensic investigators, the Keychain is often one of the most valuable sources of evidence because it contains credentials, authentication tokens, cryptographic material, and other highly sensitive information. At the same time, it is among the most heavily protected components of iOS. Successfully acquiring Keychain data often depends on factors such as device state, whether the device is unlocked, the availability of trusted pairing records, and the acquisition method used. Understanding these protections allows you to better plan acquisitions and maximize the likelihood of recovering valuable evidence.

Understanding how iOS protects sensitive data is only one part of a successful forensic investigation. You must also understand how and where that data is physically organized on the device. The next section examines the Apple file system (APFS), its container-based architecture, and the filesystem structures that determine where forensic artifacts are stored and how they can be recovered.

Apple file system (APFS)

The file system determines how data is organized, stored, and retrieved on a storage device. Just as a filing cabinet organizes documents into folders, a file system organizes every file, photograph, message, application, and database stored on a device. Understanding the file system is therefore essential for forensic investigators, because it determines where digital evidence resides and how it can be recovered.

Apple introduced the **Apple file system** (**APFS**) as the default file system for iOS devices beginning with iOS 10.3, replacing the older **hierarchical file system plus** (**HFS+**). Unlike HFS+, which was originally designed decades ago for traditional hard drives, APFS was built specifically for modern solid-state storage (SSD), strong encryption, improved reliability, and better performance. It also introduces several architectural changes that significantly affect how forensic investigators acquire and analyze evidence. Although APFS was introduced several years ago, it remains the filesystem used by current iOS 26 devices and continues to underpin Apple's modern Data Protection architecture.

One of the biggest differences in APFS is how storage is organized. Older file systems typically divided a storage device into fixed partitions, each with a predetermined size. APFS instead creates a single container that manages the available storage space. Within this container are one or more volumes, which function like independent file systems but dynamically share the same pool of storage. Instead of reserving a fixed amount of space for each volume, APFS automatically allocates storage where it is needed. As a result, volumes can grow or shrink without requiring manual repartitioning.

The following figure illustrates this architecture:

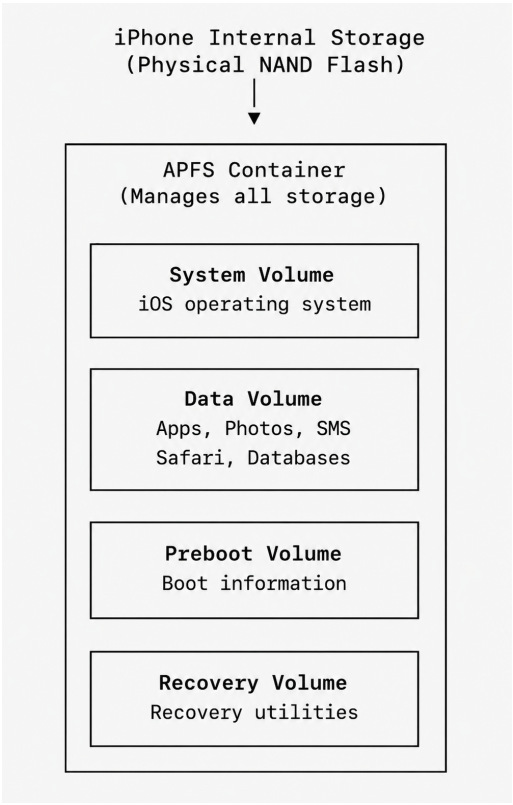


Figure 2.2: High-level architecture of an APFS container

The physical storage inside the device contains an **APFS Container**, which in turn holds multiple logical volumes. On a typical iPhone, these include a **System Volume** containing the operating system, a **Data Volume** containing user files and application data, and additional **Preboot Volume** and **Recovery Volume** used during the startup and recovery process. Although users rarely interact with these volumes directly, understanding their purpose is important because forensic artifacts may reside in different locations depending on the type of data being examined.

For forensic investigators, the data volume is usually the most significant because it contains most user-generated artifacts, including application databases, messages, photographs, documents, browser data, and application containers. However, the separation between system and user data also improves system integrity and security, meaning forensic tools must understand the APFS container structure and correctly identify each volume before evidence can be extracted and interpreted.

Key differences from HFS+

Compared to HFS+, APFS is not just an incremental improvement; it represents a fundamental rethinking of filesystem structure and behavior. HFS+ was not designed with modern encryption or SSDs in mind, while APFS includes encryption as a native, first-class feature. On iOS devices, APFS works with Apple's data protection architecture, where files are protected using per-file keys that participate in a broader hierarchy of class keys, passcode-derived keys, and device-specific hardware keys.

APFS also replaces the journaling system used in HFS+ with a copy-on-write mechanism. Rather than modifying files or metadata in place, APFS writes changes to new locations and updates references. This improves data integrity and crash protection but complicates forensic recovery, as traditional "undelete" techniques based on recovering overwritten sectors are no longer effective.

Another key difference is metadata handling. APFS uses a new metadata structure that enables faster directory traversal and more efficient space management. However, this also means that tools built for HFS+ are not compatible with APFS volumes and cannot be relied upon for forensic purposes. You must use tools with full APFS support and, ideally, those that can interpret both logical file system layout and low-level volume structures.

Tip

Do not assume that a deeper acquisition automatically means that deleted files can be recovered. APFS, file-based encryption, and modern flash-storage behavior can significantly limit traditional deleted-file recovery. Deleted evidence may instead survive within application databases, WAL files, caches, logs, backups, or APFS snapshots.

Important directories in iOS

Understanding the iOS directory structure is essential for effective forensic analysis. Unlike traditional desktop systems, iOS enforces strict application sandboxing, where each app's data is isolated from others. Most user and app-generated data resides within the `/private/var` directory, which is mounted from the data volume at boot.

The subdirectory `/private/var/mobile` is of particular forensic interest. This path serves as the home directory for the default user account (UID 501, typically named "mobile"). Within it, one finds crucial folders such as the following:

- **Documents**: Often used by apps to store user-created files
- **Library**: Stores preferences, cached data, cookies, and app-specific configuration files

- **Containers and application support**: Store sandboxed app data and shared app group data
- **Keychains**: May contain the encrypted database with app secrets or authentication tokens

Another directory of interest is `/private/var/db`, which houses critical system databases, including those for Wi-Fi configurations, device provisioning, and activation records. The `/private/var/log` directory can also be valuable, offering diagnostic logs, crash reports, and system logs that provide insight into device activity and app behavior.

Finally, you should note that symbolic links may obscure the true file locations. For example, `/var` is a symbolic link to `/private/var`, and paths reported by some tools may reflect these aliases. Tools like `ls -l` on jailbroken devices or file system parsing utilities on extracted images help resolve these links accurately.

Tip

Paths displayed by forensic tools may not always match the paths observed directly in an extracted filesystem. iOS uses symbolic links and dynamically assigned application container identifiers, so validate important artifact locations against the underlying filesystem when necessary.

Snapshots and their forensic implications

One of the most powerful and unique features introduced with APFS is **snapshot** functionality. A snapshot is a point-in-time, read-only copy of the filesystem state. It captures the state of the volume at a particular moment without duplicating the entire dataset. Snapshots are lightweight because they only store changes that occur after their creation.

iOS routinely creates snapshots for system updates, and third-party apps with entitlement access can sometimes trigger them as well. These snapshots are stored within the same volume and are managed internally by the filesystem. From a forensic perspective, snapshots can serve as valuable time capsules. If a snapshot exists from a point prior to a user deleting a file or altering data, it may still be possible to recover the previous version from that snapshot.

However, access to snapshots on non-jailbroken devices is limited. Most standard forensic tools cannot read APFS snapshots unless full file system acquisition is possible. On supported devices where a sufficiently complete decrypted filesystem acquisition can be obtained, snapshots may be available for further examination. Some tools like Cellebrite and Elcomsoft support parsing APFS snapshots during full file system extraction, making them a critical part of the evidence acquisition workflow when available.

It is also important to note that snapshots are ephemeral. The operating system may purge them without notice, especially under storage pressure. Therefore, the forensic value of snapshots is often time-sensitive and acquisition must be prioritized accordingly.

> **Tip**
>
> Treat accessible APFS snapshots as potentially time-sensitive evidence. Document and preserve them as early as possible because iOS may automatically remove snapshots as storage conditions or system state change.

Forensic considerations with APFS

APFS's encryption, copy-on-write design, and snapshot features significantly impact forensic workflows. Obtaining a full or near-full APFS filesystem acquisition depends on the device model, iOS version, device state, available authentication material, and the acquisition technique supported by the forensic tool. Access to encrypted user data depends on the availability of the appropriate data protection keys, which are tied to device-specific hardware and, for many protection classes, the user's passcode and device state.

You must ensure that your tools can interpret APFS metadata, extract meaningful artifacts from the `/private/var` subtree, and recognize and mount all volumes within an APFS container. Snapshots, when accessible, should be carefully documented, extracted, and analyzed, particularly when trying to establish a timeline of events or recover deleted content.

Understanding APFS is essential because it determines where evidence is stored and how it can be recovered. However, a forensic examiner's success depends not only on the filesystem but also on the security mechanisms that protect it. The next section examines how Apple's continuing security enhancements have changed the landscape of iOS forensic investigations and influenced the techniques available to investigators.

Impact of modern security enhancements on forensics

As Apple has steadily hardened the iOS ecosystem, each iteration of the platform has introduced new security mechanisms that directly affect the viability, depth, and ease of forensic investigations. These enhancements, while beneficial for user privacy and device integrity, have become significant barriers for forensic examiners, especially when attempting low-level data acquisition or password recovery.

File system encryption vs. full disk encryption

One of the key distinctions in Apple's security architecture is the use of **file-based encryption** (**FBE**), as opposed to traditional **full-disk encryption** (**FDE**). In iOS, every file is encrypted using a unique key, and these file keys are, in turn, protected by class keys derived from the user's passcode and hardware-based keys stored in the Secure Enclave. This granular encryption model means that even if an examiner gains access to the filesystem, individual files may remain inaccessible unless the appropriate class keys are available.

Unlike traditional full-disk encryption, where unlocking the device exposes the entire filesystem, iOS controls access on a per-file basis using data protection classes. For example, files assigned to the complete protection class remain encrypted even after boot, until the user unlocks the device after a reboot. This distinction gives rise to two important forensic states:

- **Before first unlock** (**BFU**): After a device has been powered on or restarted, but before the user enters the passcode for the first time, the device is in the BFU state. During this period, many data protection class keys are unavailable, and access to user data is significantly restricted. Although certain system files and metadata may still be accessible, most user-generated content remains encrypted.
- **After first unlock** (**AFU**): Once the user successfully unlocks the device after boot, it enters the AFU state. Additional data protection class keys are available, allowing many protected files to be decrypted. Even if the device is subsequently locked, many of these keys remain available until the next reboot, enabling forensic tools to access substantially more evidence than is possible in the BFU state.

Forensic investigators generally prefer to preserve devices in the AFU state whenever possible, as it provides significantly greater access to application data, databases, messages, and other user artifacts. Recent versions of iOS further strengthen security by automatically restarting devices after extended periods of inactivity, returning them to the more restrictive BFU state and reducing the amount of data available for forensic acquisition.

> **Tip**
>
> If a seized iPhone is already in the AFU state, avoid unnecessary actions that could cause it to restart. A reboot returns the device to BFU, where many Data Protection keys become unavailable and substantially less user data may be accessible.

Secure Enclave processor and passcode resistance

The **Secure Enclave processor** (**SEP**) represents a critical barrier in modern forensic acquisition. SEP is an isolated, tamper-resistant coprocessor that handles all cryptographic

operations related to data protection, including the enforcement of passcode delays and the management of encryption keys. Because the SEP is hardware-bound and its firmware is signed and encrypted, it cannot be trivially bypassed or debugged, even by those with physical access to the device.

This design has rendered traditional brute-force attacks virtually impossible. SEP enforces rate limits and escalating delays between passcode attempts, significantly restricting brute-force attacks. If the device owner has enabled the **Erase Data** option, iOS can erase cryptographic key material after ten consecutive failed passcode attempts. This prevents straightforward offline passcode guessing because possession of encrypted storage data alone does not provide the hardware-bound cryptographic material required to derive the necessary decryption keys.

Furthermore, even with access to the encrypted file system image, without the user's passcode or biometric authentication at runtime, the decryption keys required to access content remain inaccessible. This is a significant shift from earlier devices (such as iPhone 4 or some iPhone 5 models), where hardware vulnerabilities could sometimes be leveraged to bypass protections.

Secure Boot chain and imaging limitations

Apple's Secure Boot chain, also known as the **chain of trust**, ensures that every stage of the boot process (from the initial Boot ROM to the kernel) is cryptographically verified and signed by Apple. Each component checks the integrity of the next before execution, and if tampering is detected, the boot process halts. This significantly restricts the ability to run custom or unsigned code on a device, particularly on newer models for which publicly known bootloader-level exploits are unavailable.

For forensic purposes, this tight chain of trust limits the possibility of live imaging the device or booting into a custom recovery mode for memory dumps. Even tools that once leveraged **device firmware upgrade** (**DFU**) mode to load forensic ramdisks are now constrained by SEP protections and Secure Boot validation, which reject any attempt to load unsigned images. Consequently, traditional full physical acquisition is generally unavailable on modern iPhones unless a device-specific vulnerability or supported forensic technique can overcome the relevant security protections.

Tip

If one forensic tool reports that an acquisition method is unsupported, do not immediately conclude that the evidence itself is inaccessible. Evaluate alternative acquisition paths such as trusted pairing records, local backups, synchronized cloud data, application-specific sources, or another validated forensic tool.

Together, Data Protection, the Secure Enclave, and the Secure Boot chain significantly restrict traditional low-level acquisition techniques. Modern iOS investigations therefore require you to select acquisition methods based on the specific device, security state, available authentication material, and investigative objective.

Jailbreaking and its forensic implications

Jailbreaking is the process of bypassing Apple's built-in software restrictions to obtain elevated privileges on an iOS device. This is typically achieved by exploiting vulnerabilities in the Boot ROM, kernel, or other privileged components of the operating system. A successful jailbreak allows unsigned code to execute, provides privileged access to portions of the operating system, and enables capabilities that Apple normally prohibits, such as modifying system behavior, accessing protected areas of the filesystem, and installing software outside the App Store.

Historically, jailbreaking played a significant role in iOS forensic investigations because it allowed examiners to obtain filesystem access that was otherwise unavailable through Apple's standard interfaces. On supported devices, a jailbreak can enable access to application containers, system databases, log files, and other forensic artifacts that may not be available through logical acquisitions alone. This deeper level of access can substantially improve the completeness of an examination, particularly when recovering application data or examining low-level operating system artifacts.

However, the role of jailbreaking in modern iOS forensics has changed considerably. Over successive iOS releases, Apple has introduced numerous hardware and software security mechanisms, including the Secure Enclave, Secure Boot, PAC, KTRR, MIE, and increasingly sophisticated exploit mitigations. These protections have made publicly available jailbreaks for modern devices increasingly rare. As a result, contemporary forensic investigations often rely on commercial forensic solutions, trusted pairing records, backup-based acquisitions, or vendor-specific acquisition techniques rather than traditional public jailbreaks.

Even when available, jailbreaking should be approached cautiously during forensic examinations because the process modifies the device and may create files, logs, or configuration changes. You should document the original device state, the jailbreak and tool versions used, every action performed, and any known modifications introduced during the process. Where possible, less invasive evidence sources should be acquired before modifying the device.

Tip

A jailbreak is not a forensically neutral operation. Before using one, determine whether the required evidence can be obtained through a less invasive acquisition method and document why jailbreaking was necessary.

Most jailbreaks rely on exploiting vulnerabilities in privileged system components and therefore support only specific combinations of hardware and iOS versions. Broadly speaking, jailbreaks fall into two categories:

- **Hardware-based jailbreaks** exploit vulnerabilities embedded in immutable hardware components such as the Boot ROM. Since Boot ROM code cannot be updated after manufacture, these exploits remain effective for the lifetime of the affected hardware. The most notable example is the checkm8 exploit, which affects devices powered by Apple's A5 through A11 processors and forms the basis of tools such as checkra1n and palera1n.
- **Software-based jailbreaks** exploit vulnerabilities within iOS itself, typically in the kernel or other privileged components. Because these vulnerabilities can be patched through software updates, support is usually limited to specific iOS versions and often disappears once Apple releases security updates.

Many jailbreak environments also install alternative package managers such as Sileo or Zebra, allowing users to install third-party software and system modifications from unofficial repositories. While these tools are primarily intended for customization, they can also be useful in controlled forensic environments by allowing you to install forensic utilities, enable SSH access, or execute custom acquisition scripts. Conversely, the presence of jailbreak software on a seized device may indicate prior user modification and should be documented during the examination.

The following table summarizes several jailbreak tools that have been widely used in the forensic community:

Tool	Exploit type	Typical device support	Forensic relevance
checkra1n	Hardware (checkm8 Boot ROM exploit)	A7–A11 devices	Can facilitate deeper filesystem access and forensic acquisition on supported devices.
palera1n	Hardware (checkm8 Boot ROM exploit)	Supported checkm8 devices running newer iOS versions	Modern successor for checkm8-based devices with ongoing community support.
unc0ver	Software (kernel exploit)	Selected iOS versions	Legacy jailbreak used on specific firmware versions for filesystem access.
Taurine	Software (kernel exploit)	Selected iOS 14 devices	Legacy jailbreak supporting a limited range of devices and iOS versions.

Table 2.1: Common jailbreak tools and their uses

Although public jailbreaks have become increasingly uncommon on modern iPhones, they remain an important part of the history of iOS forensics and continue to provide valuable acquisition opportunities for supported devices. Understanding how jailbreaks work, when they can be safely used, and their impact on forensic integrity helps you select appropriate acquisition strategies while appreciating how Apple's evolving security architecture has reshaped the forensic landscape.

Summary

In this chapter, we explored the foundational concepts that underpin modern iOS forensics. We examined the architecture of the iOS ecosystem, the layered design of the operating system, and the hardware and software security mechanisms that protect user data. We also

discussed how technologies such as the Secure Enclave, data protection, Secure Boot, kernel integrity protections, MIE, ADP, and APFS influence what evidence can be acquired and the techniques available to forensic investigators. Understanding these components is essential because every forensic acquisition is constrained by the security architecture of the device.

The availability of evidence depends on factors such as the device model, iOS version, hardware generation, encryption state, whether the device is in the BFU or AFU state, and whether cloud-based protections such as ADP are enabled. Modern iOS releases, including iOS 26, continue to strengthen these protections, making low-level acquisition increasingly difficult while raising the importance of selecting the appropriate acquisition strategy.

We also examined how sensitive information is protected through the Keychain, iCloud Keychain, passkeys, and hardware-backed cryptography, and how APFS organizes data using containers, volumes, snapshots, and strong encryption. Finally, we discussed the changing role of jailbreaking in modern iOS forensics, where public jailbreaks have become increasingly rare and forensic investigations now rely more heavily on legitimate acquisition methods, trusted pairing records, backups, cloud evidence, and specialized commercial forensic tools.

A solid understanding of these architectural and security concepts forms the foundation of every successful iOS forensic examination. Rather than viewing forensic tools as "black boxes," investigators who understand how iOS stores, protects, and manages data are better equipped to choose appropriate acquisition methods, interpret recovered artifacts correctly, recognize the limitations imposed by Apple's security model, and defend their findings in legal proceedings.

In the next chapter, we move from theory to practice by examining the various techniques available for acquiring data from iOS devices. We will compare logical, filesystem, backup-based, cloud, and advanced acquisition methods, discuss the prerequisites and limitations of each approach, and provide practical guidance for selecting the most appropriate acquisition strategy based on the device, iOS version, security configuration, and investigative objectives.

3

Data Acquisition from iOS Devices

An iOS device recovered from a scene is often one of the richest sources of digital evidence available. Smartphones carry a wide range of personal information, including communications, location records, browsing history, app usage, cryptographic tokens, and more. They are highly personal devices that often document a user's movements, contacts, and intent in ways that few other pieces of evidence can. Because of this, proper handling and acquisition are critical. A single misstep during seizure or imaging can destroy valuable evidence or make it legally inadmissible.

Acquiring data from iOS devices is technically and legally complex. Apple's hardware and software design places strong emphasis on security and privacy, using hardware-backed keys, the Secure Enclave, data protection classes, secure boot chain, lockdown mode, passkeys, and newer runtime protections introduced in recent iOS releases, all of which limit low-level access. These measures protect users but also constrain examiners. So, as examiners, you must plan acquisitions with up-to-date knowledge of the device model, chipset generation, and iOS version to select the most appropriate, defensible method.

This chapter focuses on acquiring data directly from iOS devices. It explains how to assess device state at seizure, how to isolate and document devices to prevent evidence loss, and how to decide between logical, file system, and physical acquisition techniques based on the device's hardware and software characteristics. Practical workflows and tool-aware guidance are provided so that you can choose the safest and most effective method for each case. Legal considerations such as warrants, consent, and chain of custody are integrated throughout, because technical work must always be done within lawful bounds and recorded in a way that stands up in court.

The chapter covers the following topics:

- Operating modes of iOS devices
- iOS-specific legal and forensic considerations
- iOS isolation techniques to prevent remote wipe and data modification
- iOS logical acquisition
- iOS filesystem acquisition
- iOS physical acquisition
- iOS diagnostic and unified log acquisition
- Decision framework for selecting iOS acquisition methods

By the end of this chapter, you will be able to evaluate an iOS device at seizure, select an acquisition strategy based on the device generation and iOS version, and perform preliminary logical and filesystem acquisitions in a forensically sound way. Let's begin with various operating modes of iOS devices.

> **Note**
>
> **Download the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Operating modes of iOS devices

Before diving into acquisition techniques, it is essential to understand the **operating modes** of an iOS device. Different forensic methods and tools may require a device to be placed in a specific mode, and each mode represents a different state of access and control. While many commercial forensic tools can guide you through the steps, as an examiner, you must understand what each mode means, what risks it carries, and how it affects evidence preservation.

Broadly, an iOS device can run in **normal mode**, **recovery mode**, or **device firmware update (DFU) mode**. Each mode interacts differently with the device's boot process and, in turn, affects the acquisition methods available.

Normal mode

Normal mode is the everyday operating state of an iPhone or iPad. When the device is powered on, as we covered in the previous chapter, it goes through Apple's secure boot chain, which ensures that only trusted software signed by Apple can run. Each stage verifies the next using cryptographic signatures, preventing unauthorized or tampered code from loading.

Here is a more detailed explanation of the sequence of events that occur when an iPhone boots up:

1. Boot ROM code, which is embedded in read-only memory and cannot be altered, contains the Apple root certificate authority (CA) public key. This key is used to verify the signature of the next stage before it is allowed to load.
2. When the iPhone is powered on, the application processor begins execution by running the Boot ROM code.
3. The Boot ROM then verifies and loads the low-level bootloader (LLB). On newer devices (A10+), this function is merged directly into iBoot.
4. Once the LLB finishes its tasks, it verifies and loads the second-stage bootloader known as iBoot.
5. iBoot plays a critical role: it verifies and loads the iOS kernel, ensuring that the operating system is intact and untampered.
6. Finally, the iOS kernel runs and verifies all user applications, enforcing that they too are signed and trusted before execution.

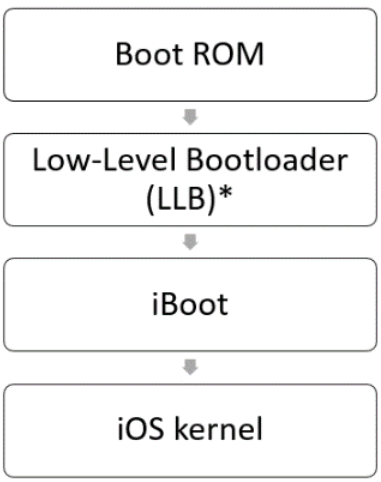


Figure 3.1: A secure boot chain of an iPhone in normal mode

In forensic practice, a device in normal mode allows for logical acquisitions using trusted tools such as Cellebrite UFED, Magnet AXIOM, or Oxygen Forensic Detective. This means data such as contacts, call logs, messages, and app data can be extracted through standard APIs, provided you have the proper authorization and credentials.

Note

In earlier generations of iPhones, the LLB was a distinct stage of the secure boot chain. The Boot ROM (immutable code burned into silicon) would verify and load the LLB, which then verified and loaded iBoot.

Starting with the A10 chip and later, Apple simplified the boot chain. The LLB stage was removed, and now the Boot ROM verifies and loads iBoot directly. In other words, LLB's role was folded into the secure boot process to reduce complexity and potential attack surfaces.

For devices that are locked, however, normal mode presents significant challenges due to Apple's hardware-backed encryption and Secure Enclave. Without a passcode, access to sensitive data is extremely limited.

Recovery mode

Recovery mode is a special boot mode that allows firmware updates and device restoration when the normal operating system cannot be started or when a restore is required. It can be entered manually by the user or invoked automatically in certain failure scenarios. In this state, the iPhone or iPad displays the familiar "Connect to computer" screen (previously "Connect to iTunes"), as shown in the following figure:



Figure 3.2: iPhone connected in recovery mode

Recovery mode is primarily used to perform upgrades, restores, or troubleshooting when the normal boot process cannot complete. From a forensic standpoint, it is important to understand this mode, as some acquisition techniques may require or automatically place the device into recovery mode.

Entering recovery mode

The procedure for entering recovery mode depends on the device generation. On modern iPhones running iOS 26, the following steps are used:

1. Connect the iPhone to a computer using a USB cable
2. Quickly press and release the **volume up** button
3. Then, press and release the **volume down** button
4. 4. Press and hold the **side** button until the recovery mode screen appears

Older iPhone models use different button combinations to enter recovery mode. Although these devices are encountered less frequently in modern forensic investigations, the procedures remain available in Apple's documentation for legacy hardware.

Exiting recovery mode

On modern iPhones, including those running iOS 26, you can usually exit recovery mode by restarting the device:

1. Quickly press and release the **volume up** button
2. Then, press and release the **volume down** button
3. Press and hold the **side** button until the Apple logo appears

Older iPhone models use different button combinations to restart the device from recovery mode. Although these devices are encountered less frequently in modern forensic investigations, the procedures remain available in Apple's documentation for legacy hardware.

Recovery mode provides a controlled environment that allows forensic tools to communicate with the device before the operating system fully loads. Depending on the device model, iOS version, and acquisition technique, this may enable certain low-level forensic operations, such as loading forensic ramdisks on supported devices, collecting device metadata, or performing specialized acquisition procedures. However, on modern iPhones, recovery mode alone does not bypass Apple's security protections or provide unrestricted access to encrypted user data.

Note

Recovery loops:

Sometimes, an iPhone may become stuck in a continuous recovery mode cycle, known as a recovery loop. This often occurs after a failed jailbreak attempt, incomplete firmware update, or corruption of system files. Breaking out of this loop usually requires restoring the device through iTunes or Finder.

DFU mode

Device firmware upgrade (DFU) mode is a special, low-level state of an iOS device that bypasses the normal boot chain and recovery processes. Unlike recovery mode, which still relies on iBoot, DFU mode allows the device to communicate directly with iTunes/Finder (or forensic tools) without loading iBoot or iOS. This makes it useful for advanced troubleshooting, firmware restoration, and in certain cases for forensic acquisition.

Tip

Restoring the device will overwrite evidence, so you should first test and validate extraction methods on non-evidentiary devices before attempting a live case.

During a normal or recovery mode boot, the Boot ROM verifies and loads iBoot, Apple's second-stage bootloader. In DFU mode, however, the Boot ROM does not automatically load iBoot. Instead, it enters a minimal recovery state and waits for instructions over USB. From this state, a connected computer (using Finder, Apple Devices, or specialized forensic tools) can communicate directly with the Boot ROM and transmit a signed iBoot image or other recovery components.

If the transmitted code is correctly signed by Apple, the Boot ROM verifies it and the normal boot process continues. If a Boot ROM vulnerability, such as checkm8 (A5-A11 devices) or usbliter8 (A12-A13 devices), can be exploited, investigators may gain low-level control of the boot process before iOS loads.

This early point of execution makes DFU mode particularly valuable for forensic research and advanced acquisition techniques. However, Boot ROM exploit alone does not bypass hardware-backed encryption or the Secure Enclave, and additional techniques are generally required to access protected user data.

DFU mode is normally entered manually using a specific hardware button sequence. Unlike recovery mode, DFU does not automatically appear during normal boot failures. You can manually place a device into DFU mode for analysis or reinstallation purposes. The screen remains completely black (unlike recovery mode, which shows the cable-to-computer logo).

Entering DFU mode

The procedure for entering DFU mode varies depending on the device generation. On modern iPhones, including the current iPhone 17 lineup, follow these steps:

1. Connect the iPhone to your forensic workstation using a USB cable
2. Quickly press and release the **volume up** button
3. Then press and release the **volume down** button

4. Press and hold the **side** button until the screen turns black
5. Now holding the **side** button, press and hold the **volume down** button for approximately 5 seconds
6. Release the **side** button while continuing to hold the **volume down** button for another 5 to 10 seconds

If the procedure is successful, the display will remain completely black. If the Apple logo or the recovery mode screen appears, the timing was incorrect and the process should be repeated.

Older iPhone models use different button combinations to enter DFU mode. Although these devices are encountered less frequently in modern forensic investigations, the procedures remain available in Apple's documentation for legacy hardware.

Verifying DFU mode

Since the screen is black, verification must be done on the connected workstation. For example, on a Mac workstation, you will see a message automatically as shown in the following figure:

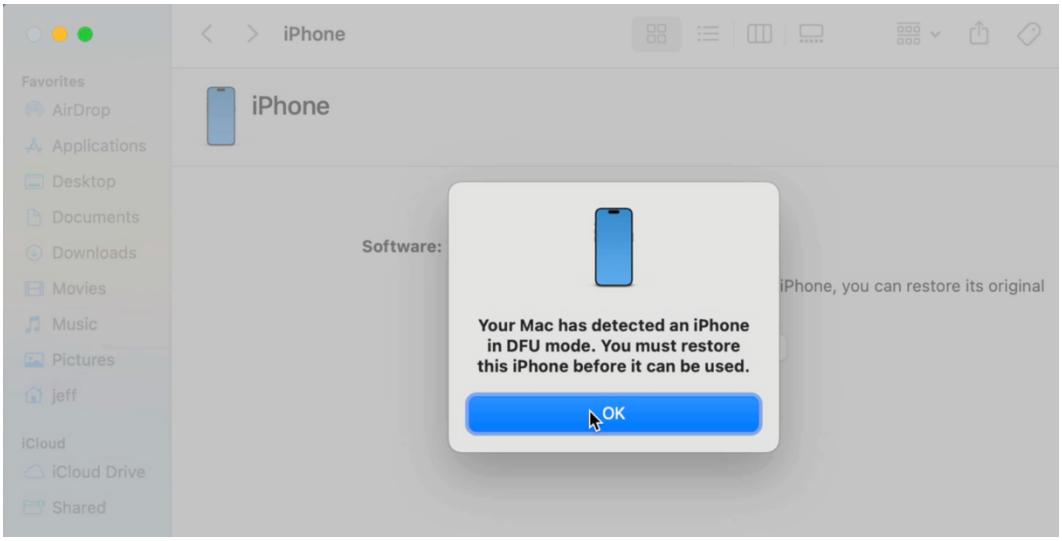


Figure 3.3: MacBook system information displaying a device in DFU mode

Since iBoot is skipped, DFU allows direct interaction with the Boot ROM, which can be exploited for low-level forensic access. For example, the checkm8 exploit leverages DFU mode on vulnerable devices (A5–A11) to enable filesystem extraction.

The next section examines some of the iOS-specific legal and forensic considerations to keep in mind.

iOS-specific legal and forensic considerations

When dealing with iOS devices at the time of seizure and acquisition, you must go beyond the generic legal and forensic principles outlined in *Chapter 1*. Apple's strong security ecosystem, frequent updates, and privacy-focused design make iPhones and iPads some of the most challenging devices to acquire evidence from. This section highlights the special considerations that apply specifically to iOS devices.

Passcodes, encryption, and the Secure Enclave

Modern iPhones and iPads running iOS 26 continue to rely on the **Secure Enclave Processor** (**SEP**) as the foundation of Apple's hardware-backed security architecture. The SEP controls the keys that lock and unlock data, and these keys are linked both to the phone's unique hardware and to the user's passcode. What this means in practice is that if an iPhone is turned off and locked, there is no way to read the data without entering the correct passcode.

> **Note**
>
> In older iPhones, you could sometimes guess the passcode quickly by trying many combinations. But on newer devices, Apple has built-in protections: after each wrong attempt, the phone forces you to wait longer before trying again, and in some cases, the device can even erase itself after 10 failed attempts if the user has enabled that option.

From a legal perspective, the type of unlock mechanism matters. In some jurisdictions (for example, under U.S. law), compelling a suspect to disclose a passcode may be protected by the Fifth Amendment as "testimonial evidence." Once an iPhone has rebooted or entered a secure lock state, biometrics are disabled until the passcode is entered. You must act quickly, and always within the bounds of legal authority.

Jailbreaking, exploits, and legal boundaries

Many acquisition techniques for iOS devices rely on exploiting vulnerabilities. Publicly available jailbreak tools, semi-tethered exploits like checkra1n (based on the checkm8 vulnerability), and proprietary vendor solutions often depend on bypassing Apple's security mechanisms. While these methods can provide invaluable forensic access, they raise potential legal and evidentiary questions.

Courts may ask: "*Was the method tested, validated, and documented? Did it alter the device or introduce doubt about evidence integrity?*" Since jailbreaks often modify the system partition or disable security features, their use must be carefully justified and fully logged. In some regions,

use of certain exploit-based tools could even raise challenges about whether the evidence was obtained lawfully.

Legal authority and warrants in the iOS context

Because of the heightened privacy controls around iOS devices, you must ensure the scope of their legal authority is clearly defined. A warrant should explicitly state whether the seizure covers the following:

- The physical iOS device
- Local data stored on the device
- Associated iCloud data (which is often stored outside the device itself)
- Application-specific data (such as WhatsApp or Signal messages)

Courts are increasingly scrutinizing overbroad warrants that request "all data" from a smartphone. As forensic examiners, you must align the acquisition with the warrant's scope and avoid going beyond it. Failure to do so can make the evidence inadmissible.

Documentation and chain of custody

As with all digital forensics, meticulous documentation is essential. With iOS, however, it becomes even more important because many acquisition steps involve proprietary tools or exploit-based methods that may be challenged in court. To avoid this, you should ensure you do the following:

- Record the device's state at seizure (on/off, locked/unlocked, connected/disconnected)
- Log all steps taken to isolate and acquire the device
- Document which tool was used, its version, and whether any known modifications were made to the device
- Retain detailed logs of exploit-based acquisitions (such as checkm8 usage) to show reliability and repeatability

Careful documentation and a well-maintained chain of custody help ensure that forensic findings remain reliable, repeatable, and legally defensible. With these administrative controls in place, the next priority is protecting the device itself from external communication that could alter or destroy potential evidence.

iOS isolation techniques to prevent remote wipe and data modification

As discussed earlier, iOS devices are uniquely vulnerable to remote commands and background activity that can alter or destroy digital evidence. Proper isolation is therefore one of the very

first technical steps after seizure. This section focuses on the practical, iOS-specific strategies examiners use to prevent unwanted changes.

Remote wipe and activation lock

Apple's Find My iPhone service, active on any device linked to an Apple ID, allows the owner to remotely issue an `erase` command. If the phone is connected to Wi-Fi or cellular service, the wipe can execute within seconds. Once completed, the device enters **activation lock**, requiring Apple ID credentials before it can be reactivated. From a forensic perspective, failure to isolate a device in time can result in permanent loss of locally stored evidence.

Even without a deliberate `erase` command, iOS devices continuously exchange data with iCloud and other Apple services. This includes syncing iMessage conversations, iCloud Photos, Health records, location logs, and application data. Push notifications and system updates can also alter timestamps and metadata. For this reason, isolation protects not only against malicious intent but also against routine background processes that could unintentionally compromise evidentiary integrity.

Effective isolation strategies

When a device is seized in an unlocked state, you should immediately enable airplane mode, disable Wi-Fi and Bluetooth through the `Settings` app (not just control center), and turn off location services if possible. Although this behavior was introduced in iOS 11, it continues in iOS 26, where control center disconnects Wi-Fi and Bluetooth from current connections without fully disabling the radios when you turn them "off" from the control center. This allows the device to continue using features like AirDrop, Apple Watch connections, AirPlay, Handoff, and location services.

For forensic purposes, that's a big problem because it means the device can still reconnect to known Wi-Fi networks automatically, maintain Bluetooth connections, and receive remote commands like a wipe request. The only way to fully disable Wi-Fi and Bluetooth is through the `Settings` app (**Settings** | **Wi-Fi** | **Off** and **Settings** | **Bluetooth** | **Off**.) That's why best practice is to use airplane mode and confirm radios are disabled in Settings, not just rely on control center.

For fieldwork, Faraday bags provide a quick and portable solution, but they are prone to leakage and cause accelerated battery drain as the device continually searches for a signal. In lab environments, more robust options such as RF shield boxes, cabinets, or tents are preferred. These allow you to connect power and USB cables while maintaining complete RF isolation, ensuring the device can be acquired without exposure to external signals.

Power and battery considerations

Battery management is particularly important. If a device powers off during isolation and requires a passcode on reboot, critical access may be lost. Shielded environments should support safe charging so that the device remains powered throughout the examination process. You should always assess charge level during seizure and maintain adequate power during transport and acquisition.

Improper isolation can have severe consequences, from total data loss due to remote `erase`, to subtle but legally damaging changes caused by background synchronization. Correctly isolating an iOS device ensures that the evidence preserved truly reflects its state at the time of seizure, which strengthens both forensic reliability and legal admissibility.

iOS logical acquisition

Logical acquisition is one of the most common starting points for forensic analysis of iOS devices. It uses Apple's own services to capture data that is accessible to the user, such as app content, call logs, messages, photos, and configuration files. Because this process relies on supported APIs, it is generally fast, reliable, and non-invasive, making it a useful first step in almost any case. However, you must understand both its strengths and its limitations.

A logical acquisition produces a dataset similar to what is included in Finder (macOS) or Apple Devices (Windows) backup. This covers contacts, SMS and iMessage databases, Safari history, photos, app directories, and many system settings. While it does not retrieve deleted files directly, SQLite databases often contain remnants of deleted records. You can later recover these records during analysis, extending the usefulness of a logical image.

Logical acquisitions are structured and easy to parse, which makes them an efficient way to preview evidence and determine whether deeper extraction is required. For example, chat histories or call records obtained through a logical backup may be sufficient in cases involving harassment or fraud without the need for physical imaging.

Prerequisites and limitations

Logical acquisition requires that the device either be unlocked or that the examiner has access to a trusted "pairing record". A **pairing record** (sometimes called a **lockdown file**) is a small file that macOS or Windows creates the first time an iOS device is connected and trusted. It contains cryptographic keys that allow that computer to communicate with the iPhone or iPad without re-entering the passcode. If the device is locked and the passcode is unknown, but a pairing record is recovered from the suspect's computer, you may still be able to initiate a backup and acquire data.

This makes pairing records an important forensic artifact in themselves. During seizure, you should consider collecting any computers the device may have synced with, since their pairing records could provide a lawful path to accessing the mobile data when the device itself cannot be unlocked. Recent versions of iOS, including iOS 26, continue to rely on trusted pairing records for authorized communication with forensic workstations, making these records valuable forensic artifacts when recovered from trusted computers.

Another limitation is that logical backups exclude certain sensitive categories of data unless backup encryption is enabled. Unencrypted backups omit Keychain content such as stored Wi-Fi credentials and email passwords. Encrypted backups, by contrast, include portions of the Keychain, making them much more valuable to investigators.

You may need to obtain the backup password from the user, attempt recovery through password attacks, or leverage forensic tools that can assist in handling encrypted backups. If the backup password is unknown, recent versions of iOS allow users to reset it by resetting all device settings. However, this process does not decrypt previously created encrypted backups, and certain device settings, such as saved Wi-Fi passwords, display settings, wallpaper, and other preferences, are reset. You should therefore carefully consider the evidentiary impact before recommending or performing this action.

Tools for logical acquisition

A wide range of forensic tools support logical acquisition of iOS devices. Industry-standard suites such as Cellebrite UFED, Magnet AXIOM, Oxygen Forensic Detective, and Elcomsoft iOS Forensic Toolkit provide guided workflows and strong reporting features. Open-source frameworks such as libimobiledevice also support logical backups and can be useful for custom scripting or in resource-constrained environments.

Each tool may extract and present data differently, and some may parse specific app artifacts more effectively than others. For this reason, it is best practice to use more than one tool and compare results. This cross-validation improves confidence in the accuracy and completeness of the data.

Forensic value and best practices

Although logical acquisition is often seen as "lightweight" compared to physical or filesystem imaging, it has significant forensic value. Because it is non-invasive, repeatable, and legally defensible, it can provide a reliable foundation for an investigation. You should always generate cryptographic hashes of backup files to validate integrity, carefully document the tool and version used, and record the state of the device at the time of acquisition.

Logical acquisition should not be treated as a replacement for more advanced techniques, but as a complementary step. In many cases, it provides fast access to user-facing data that can immediately advance an investigation. In others, it serves as a roadmap, guiding you toward deeper acquisition methods when deleted or encrypted content is critical.

Logical acquisition in practice

When conducting logical acquisition, examiners typically proceed in the following way:

- **Establishing trust**: If the device is unlocked, confirm whether the workstation is already trusted. If not, you must interact with the device to approve the connection. This may not be possible if the device is locked.
- **Initiating backup**: Use forensic software to trigger a logical backup. Many tools allow you to choose between full or partial backups, targeting specific data categories.
- **Handling encryption**: If backups are password-protected, tools like Elcomsoft Password Recovery can attempt to recover the password through brute force or GPU-assisted attacks.
- **Validating results**: Compare extracted logical data against what is visible on the device and ensure integrity using cryptographic hashes.

These steps provide a general overview of the logical acquisition process. In practice, the exact workflow varies depending on the forensic tool being used, but the underlying principles remain the same. The following walkthrough demonstrates how a typical logical acquisition is performed using a commercial forensic tool.

Practical walkthrough: Logical acquisition using a commercial forensic tool

In this example, we will perform logical acquisition of an iPhone using the Cellebrite UFED tool by following these steps:

1. Launch **Cellebrite UFED** and choose **Mobile device** from the main screen:

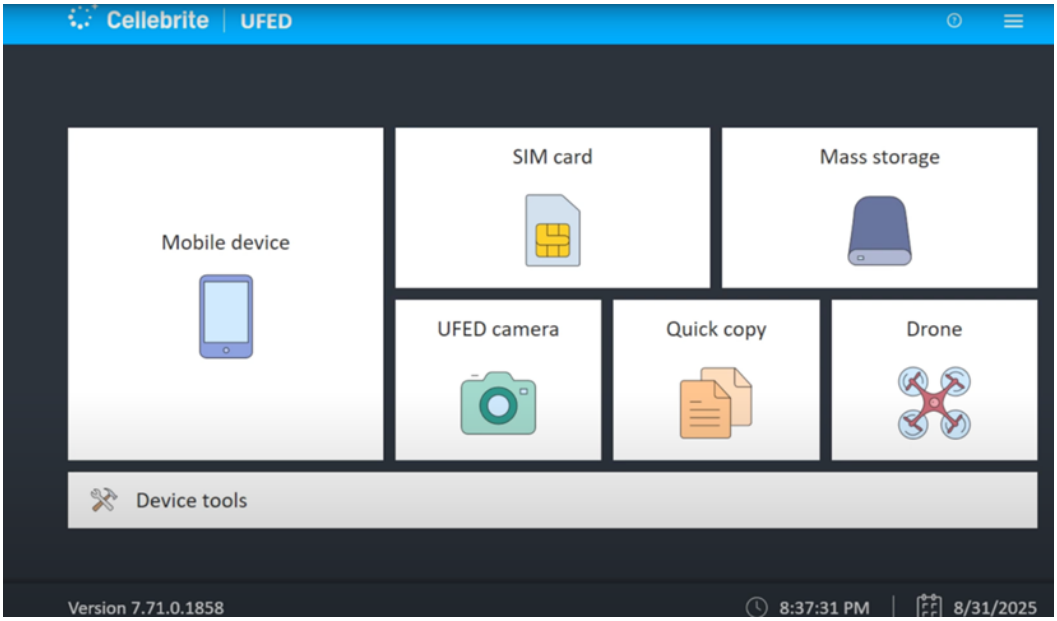


Figure 3.4: Cellebrite UFED home screen showing the available acquisition and evidence source options

2. Next, connect the device to the system running Cellebrite and select the **AUTO DETECT** option:

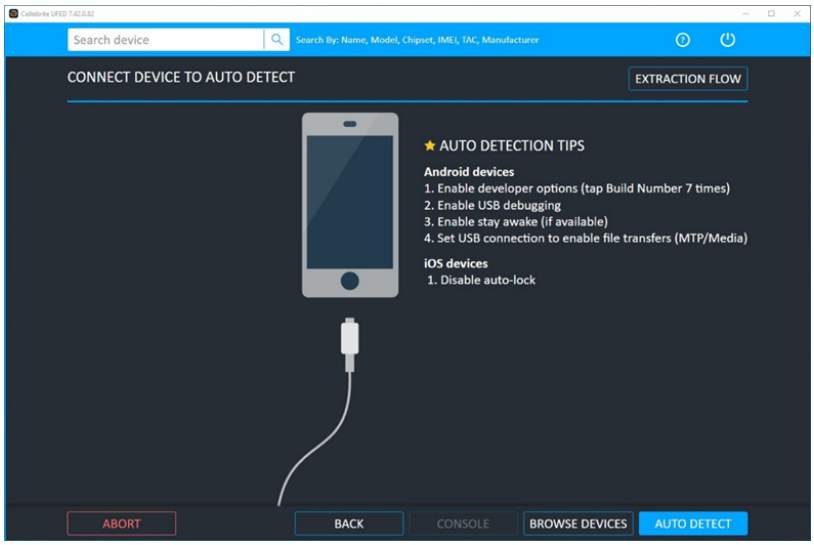


Figure 3.5: Auto Detect screen in Cellebrite UFED used for device identification

3. Once the device is detected, verify that the displayed information (such as the device model and connection status) matches the device under examination:

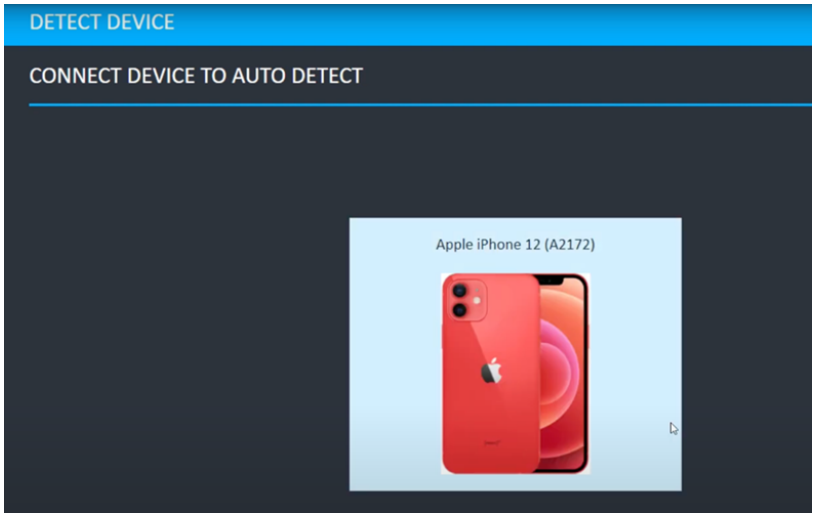


Figure 3.6: Connected iPhone successfully detected by Cellebrite UFED

4. Click on the device. You will now see available acquisition methods depending on the iPhone model. Choose the **Advanced Logical** acquisition and further on **filesystem** to continue:

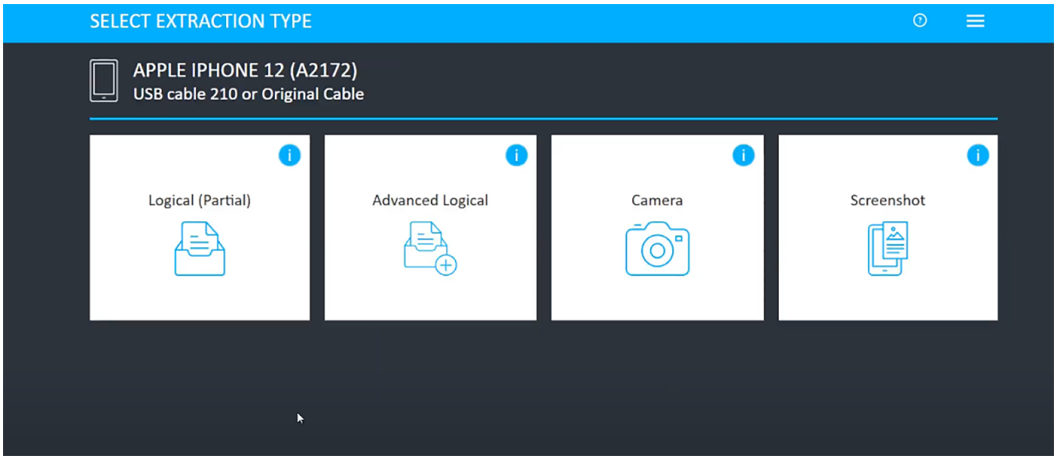


Figure 3.7: Available acquisition methods displayed after device detection

5. Enter the path where you want the files to be extracted and hit next. To perform a logical acquisition, the device must be unlocked. Consider disabling auto-lock from the device's settings. You should also trust the computer when the device prompts you for the pairing request. Allow some time for the extraction process to be completed:

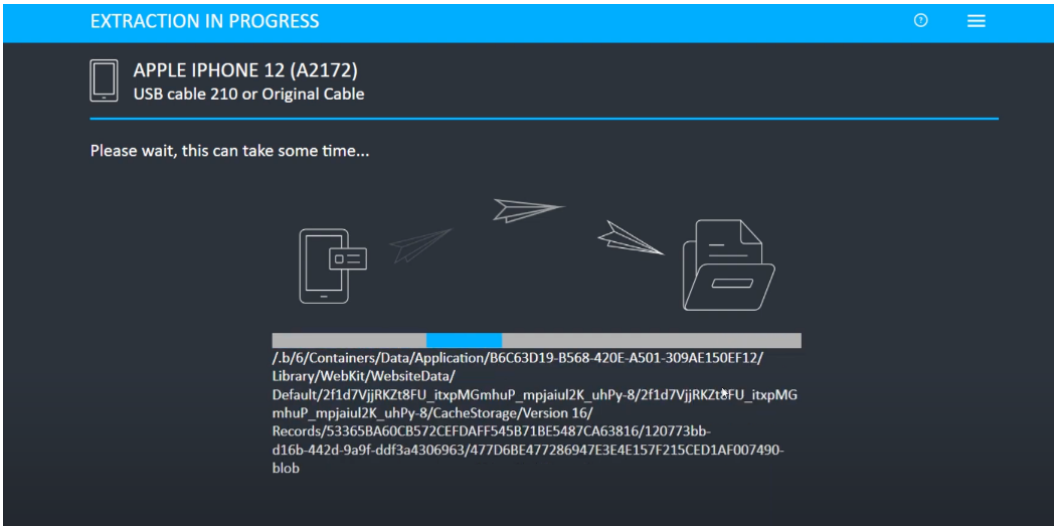


Figure 3.8: Logical acquisition in progress using Cellebrite UFED

6. Once successful, you would see a message as shown in *Figure 3.9*, confirming the same:

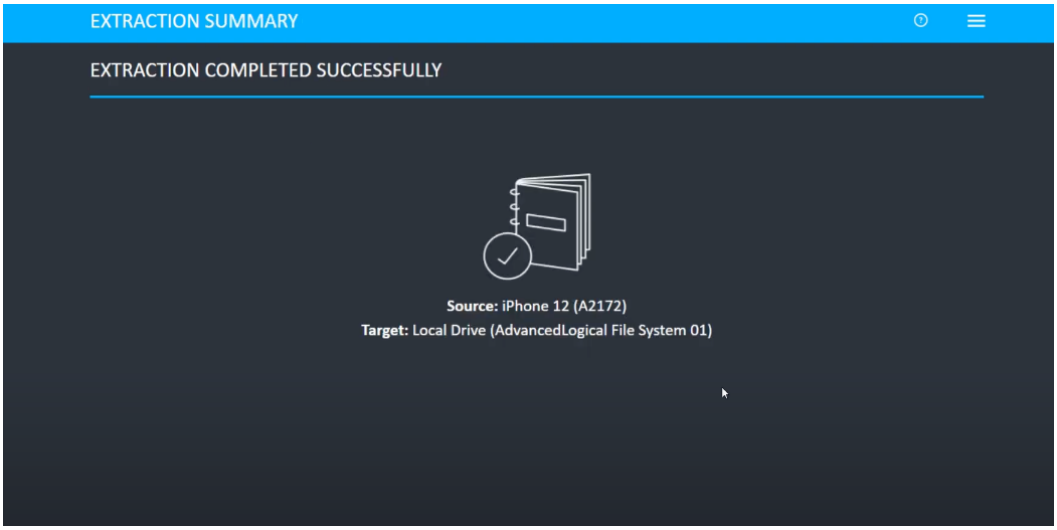


Figure 3.9: Successful logical acquisition of the iPhone using Cellebrite UFED

You can now load the acquired data into a forensic analysis tool, such as Cellebrite Physical Analyzer, for examination. Depending on the acquisition method and the device state, the extracted evidence may include contacts, call logs, SMS and iMessage conversations, photos and videos, Safari history, application data, system logs, and other user-generated artifacts. The analysis of these artifacts is covered in detail in the upcoming chapters.

iOS filesystem acquisition

Filesystem acquisition goes deeper than logical extraction. Instead of collecting only the files and records that Apple exposes through its backup APIs, filesystem acquisition attempts to capture the device's directory tree, app containers, system databases, logs, caches, and other artifacts that live on the data volume.

Note

In iOS, storage is organized into volumes under the **Apple Filesystem** (**APFS**) structure. Think of a volume as a separate "partition" or "section" of the storage, each with its own purpose.

The "data volume" is the part of the filesystem where user data and app data live. This includes app containers (each app's sandbox directory with documents, caches, and databases), system databases (like SMS, call history, contacts), and user files such as photos, videos, downloads, and so on.

> By contrast, another important APFS volume is the "system volume," which contains the core iOS operating system itself. Since iOS 15, the system volume is sealed and read-only, meaning it cannot be modified under normal circumstances.

When successful, filesystem extraction yields far richer evidence: complete application sandboxes, system configuration, crash logs, cached web content, and often traces that do not appear in a logical backup. For many investigations (serious criminal matters, insider threat or IP theft probes, and advanced incident response), filesystem access is essential. Achieving it on iOS, however, is substantially more difficult than on many other platforms because of Apple's layered security model. The following subsections explain the most common approaches, their benefits, and their operational and legal tradeoffs.

What makes filesystem acquisition different and valuable

Filesystem acquisition recovers entire directories and files in their on-disk layout. This supports more thorough timeline reconstruction and enables recovery of artifacts that logical backups cannot offer. Examples include application caches, system logs in `/private/var`, Safari's full WebKit cache, databases in app-specific containers, and filesystem metadata such as timestamps and extended attributes. Filesystem images also facilitate forensic techniques such as carving, journal/WAL analysis for deleted content, snapshot comparison, and deeper corruption analysis. Because iOS employs copy-on-write and snapshots, filesystem extraction that preserves snapshot metadata can reveal previous versions of files that are no longer present in the live filesystem.

Before trying filesystem acquisition, you must check both the condition of the device and the legal authority for the work. In most cases, you need the device to be unlocked, a trusted pairing record, or an exploit that gives higher-level access. On newer iPhones with the Secure Enclave and a secure boot chain, low-level access is often impossible unless a special vendor tool or hardware exploit is used. It is also essential to confirm that the search warrant or user consent clearly allows the level of device modification required, and to record this authorization before moving forward.

Unlike logical methods, filesystem acquisition typically requires some level of elevated access to the device. Depending on the iPhone model and iOS version, this can be achieved through an agent-based method (where a trusted forensic app temporarily runs on the device) or through jailbreak-assisted methods (where system protections are bypassed using vulnerabilities). Both approaches come with benefits and risks, which we will explore in the following sections.

Agent-based acquisition

Agent-based acquisition is a pragmatic middle ground that many commercial vendors use. In this approach, a signed, temporary agent or extraction application is installed on the target device and executed to enumerate and copy files from the device's filesystem to the examiner's workstation.

An agent normally requires an unlocked device and an authorized pairing. The agent is code-signed and presented to the device as a legitimate application; when run, it often leverages elevated (but still user-space) permissions to read application containers and system directories that are accessible from the user session. The agent contains small programs that scan folders, copy app databases, grab photos and videos, and collect system log files. Results are exported either as an archive or streamed directly to the forensic workstation.

As Apple continues to strengthen kernel and runtime protections in the latest iOS versions, agent-based acquisition has become one of the preferred approaches for supported, unlocked devices because it minimizes system modification while still providing access to valuable forensic artifacts.

From a forensic perspective, agent-based acquisition offers several advantages, but it also introduces operational considerations that you should keep in mind. One important benefit is that agent-based extraction is less invasive than jailbreaking and can yield a broad range of files that logical acquisition misses. It works across many iOS versions and device models as long as the device is unlocked and allows installation of the agent. Since agents are typically signed, they do not attempt to alter the kernel or underlying protected regions and therefore are often more defensible in court than exploit-based techniques.

The following is the operational checklist for agents:

- Confirm legal authority to install and run code
- Photograph the device state
- Preserve any existing pairing records
- Use a controlled, isolated workstation
- Record tool version and agent binary hash
- After extraction, collect a directory listing of the device to document agent-created files and remove the agent only if doing so is consistent with best practices and legal requirements

Commercial suites such as Cellebrite UFED, Magnet AXIOM, and Oxygen Forensics provide agent-style acquisition modes. Their agents are vendor-signed and typically installed only for the duration of the extraction and removed afterwards. Always capture agent logs, timestamps of installation and removal, and a binary hash of the agent itself.

Jailbreak-assisted acquisition

Jailbreak-assisted acquisition aims to remove Apple-imposed restrictions to gain root access to the filesystem. In a forensic context, jailbreaks are used to elevate privileges sufficiently to read system partitions, traverse all app sandboxes, access hidden logs, and in some cases retrieve Keychain items or APFS snapshots. As covered in earlier chapters, a jailbreak typically exploits a kernel or Boot ROM vulnerability to disable code signing, enable root shell access, or boot into a privileged environment where filesystem mounts can be read.

Tools such as checkra1n (which exploits the checkm8 Boot ROM vulnerability on A5–A11 devices) have been used in forensic labs to gain broader access. Once jailbreak is achieved, you may install SSH daemons, use rsync or tar to copy directories, mount internal storage over USB, or run specialized forensic agents that require root.

Jailbreak-assisted acquisition can provide significantly deeper access to an iOS device than logical or agent-based methods, but it also introduces important technical, legal, and forensic considerations. The following points summarize the primary benefits and limitations of this approach. A successful jailbreak can provide near-complete access to the device filesystem including directories otherwise inaccessible, system logs, crash reports, and sometimes older snapshot data. This access can reveal deleted objects, deeper app artifacts, and system-level evidence that logical or agent-based methods cannot obtain.

However, jailbreaking modifies the device at a deep level. Kernel patches, boot modifications, and changes to system integrity break the original state, which raises admissibility concerns. Expert testimony will likely be required to explain the method and to demonstrate that the data extracted is reliable and has not been inadvertently altered by the jailbreak. Additionally, jailbreaks vary by iOS version and device model, and many modern devices and iOS releases are not "jailbreakable." The permanence of some jailbreak modifications is another concern: even after un-jailbreaking, traces may remain in logs or metadata.

The following is the operational checklist for jailbreaking:

- Use jailbreaks only when necessary and authorized.
- Fully document the exploit chain, the exact binary and version used, and the steps taken.
- Where possible, test the exact jailbreak procedure on an identical lab device and capture results to show repeatability.
- Record hashes of critical files pre- and post-extraction to the extent possible and include these in court-facing documentation. If the method could alter the data of interest, openly disclose the limitations in the report.

Hardware-level exploits such as checkm8 are powerful because they cannot be patched by Apple with software updates on affected devices. They are useful for older chip families but do not apply to modern A12+ devices. Because such exploits operate at a very low level, they may allow raw memory or filesystem access without reliance on the operating system. Still, using them demands the same rigorous documentation and validation as any exploit.

Vendors approach filesystem extraction differently. Some deploy signed agents as described earlier while others offer proprietary exploit-based modules or integrated hardware solutions that automate jailbreaks and filesystem pulls. Examples include vendor-branded jailbreak/imaging modules in suites like Cellebrite, Magnet Graykey appliances used by law enforcement, and open-source tools combined with custom scripts in academic or lab settings. Evaluate vendor claims carefully and require validation artifacts such as test reports and third-party testing results before relying on a vendor's exploit in sensitive matters.

Because filesystem acquisition often requires elevated access or system modification, meticulous documentation is mandatory. Your report should include the legal authorization, device identifiers, exact tool versions and modules used, binary hashes of any agents or exploits, a step-by-step log of actions taken, timestamps, and pre- and post-acquisition hashes of the extracted files. If the acquisition required a jailbreak or exploit, include lab test results from a control device showing that the method is repeatable and did not materially alter relevant evidence.

When planning a jailbreak-assisted filesystem extraction, begin by distinguishing two broad device families because forensic options differ substantially between them:

- **A5 through A11 devices (iPhone 5s through iPhone X)**: These models remain vulnerable to the hardware-based checkm8 exploit, first disclosed in 2019. Because checkm8 is a Boot ROM vulnerability burned into silicon, it cannot be patched by software updates. This means that iPhones and iPads in this range can be jailbroken regardless of whether they run iOS 13, iOS 17, or future updates.
  Tools such as checkra1n and its successors like palera1n have continued to evolve, supporting recent iOS versions on affected devices. For forensic practitioners, this provides a reliable pathway to obtain filesystem extractions from older hardware, assuming legal authority and proper validation are in place.
- **A12 and newer devices (iPhone XS, iPhone XR, iPhone 11 and beyond)**: Beginning with the A12 chip, Apple introduced significant hardware security enhancements, including pointer authentication codes (PAC) and a hardened SEP. For several years, no public Boot ROM exploits were known for A12 and newer devices. However, in 2026, researchers publicly disclosed usbliter8, a Boot ROM exploit affecting A12- and A13-based devices. Although this represents a significant advancement in low-level security

research, it does not, by itself, provide a complete jailbreak or unrestricted access to encrypted user data.

Practical forensic acquisition on these devices continues to depend on factors such as the device model, iOS version, passcode state, Secure Enclave protections, and the capabilities of the forensic tools being used. Consequently, investigations involving A12 and newer devices typically rely on a combination of logical acquisition, agent-based or filesystem acquisition (where supported), cloud acquisition, and commercial forensic solutions, with the available techniques varying according to the specific hardware and software configuration.

Agent-based and jailbreak-assisted techniques provide deeper access to iOS devices than traditional logical acquisition, but they also require greater care to preserve forensic integrity and document every action performed. Understanding the strengths, limitations, and operational considerations of these methods helps investigators select the most appropriate acquisition strategy for a given device. The following walkthrough demonstrates how a typical agent-based filesystem acquisition is performed using a commercial forensic tool.

Practical walkthrough: Agent-based filesystem acquisition

Before beginning, ensure you have lawful authority to access the device and that the chosen agent workflow is covered by your warrant or consent. The following steps describe an agent-based filesystem extraction using a commercial tool. They assume the device is unlocked or a valid pairing record exists and that you have validated the procedure on matching test hardware.

1. Connect the iPhone via USB and open the **Oxygen Forensic Device Extractor** tool:

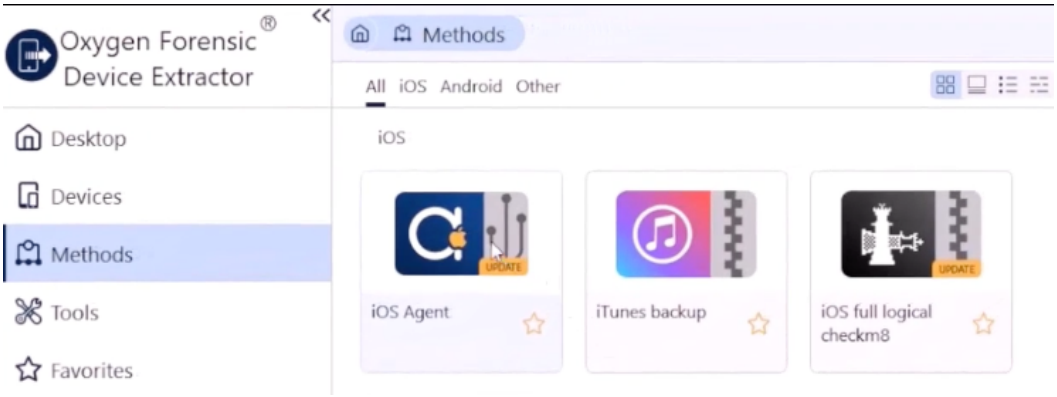


Figure 3.10: Oxygen Forensic Device Extractor showing available iOS acquisition methods

2. Under **Methods** select the **iOS Agent** option
3. Enter the path to extraction and select **Extract using iOS Agent**
4. Once the device is plugged in, Oxygen Forensics will auto-detect display-related information. Click **Start the extraction**:

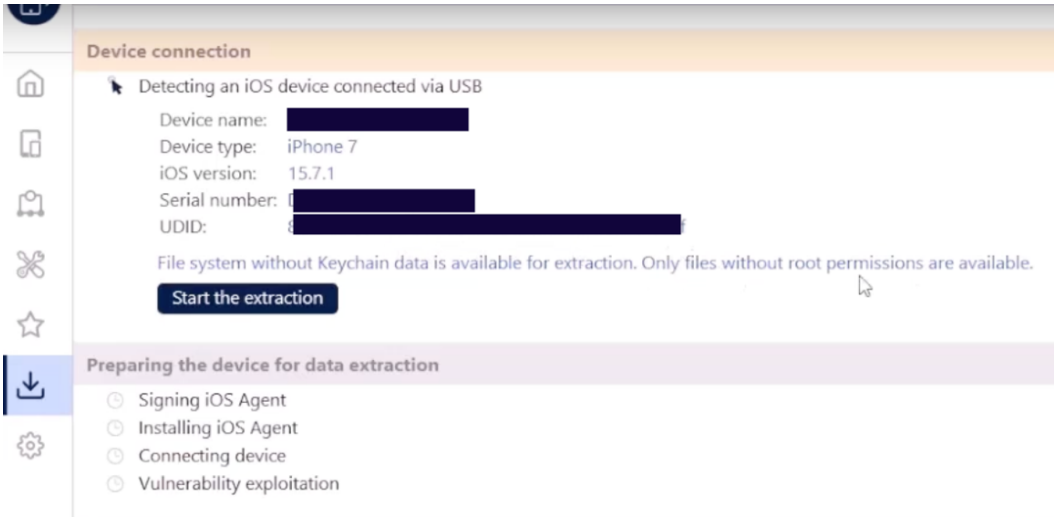


Figure 3.11: Connected iPhone detected by Oxygen Forensic Device Extractor

5. The tool prompts you to sign in to the Apple account. Once done, the tool will automatically install an iOS agent on the device.
6. On the iPhone, navigate to **Settings | General | VPN & Device Management** and you will find the iOS agent that is installed. Click **trust abc@gmail.com** option. Launch the iOS agent app on iPhone.
7. Now switch back to the computer and select **Full Extraction**. This creates a log file that contains filesystem details which can be further analyzed.

We will now look at a practical walkthrough of jailbreak-assisted acquisition.

Practical walkthrough: Jailbreak-assisted filesystem acquisition

Below is a step-by-step process to perform full filesystem acquisition of iOS devices using Cellebrite UFED:

1. Launch Cellebrite UFED and select **Mobile device** from the main menu
2. Then, choose the device model from the search option or use **Auto Detect** to detect it automatically. Keep in mind that you will need to use a USB-A cable to perform the full filesystem acquisition.
3. Select **Advanced Logical** option as shown in *Figure 3.12*:

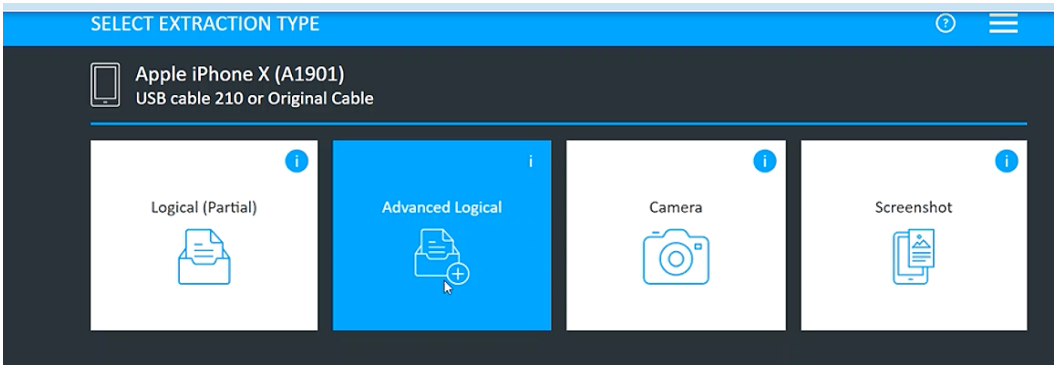


Figure 3.12: Selecting Advanced Logical acquisition in Cellebrite UFED

4. Further select the **Full File System (checkm8)** option:

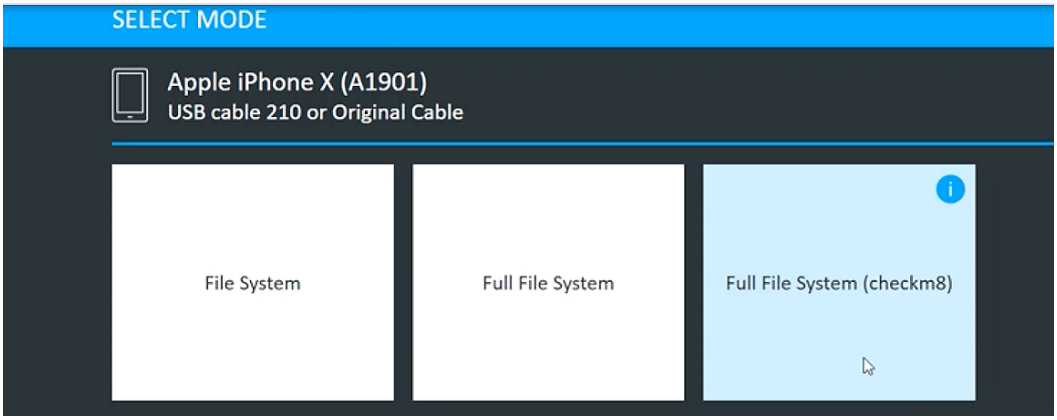


Figure 3.13: Full File System (checkm8) acquisition option in Cellebrite UFED

5. Now power off the device and enter **Recovery mode**. From there, put the device in DFU mode following the onscreen instructions. Once the device enters DFU mode, the **Continue** button will be enabled:

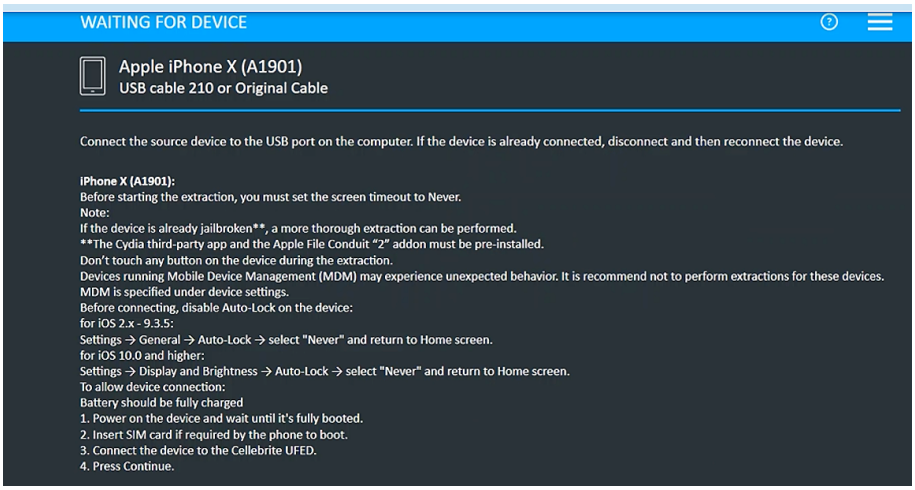


Figure 3.14: Preparing the iPhone for checkm8 acquisition by entering DFU mode

6. UFED will now exploit the device using the Checkm8 vulnerability. This process may take several minutes. All actions run in the device's volatile memory, so no permanent changes are made to the device; it will return to its normal boot state once the acquisition is completed:

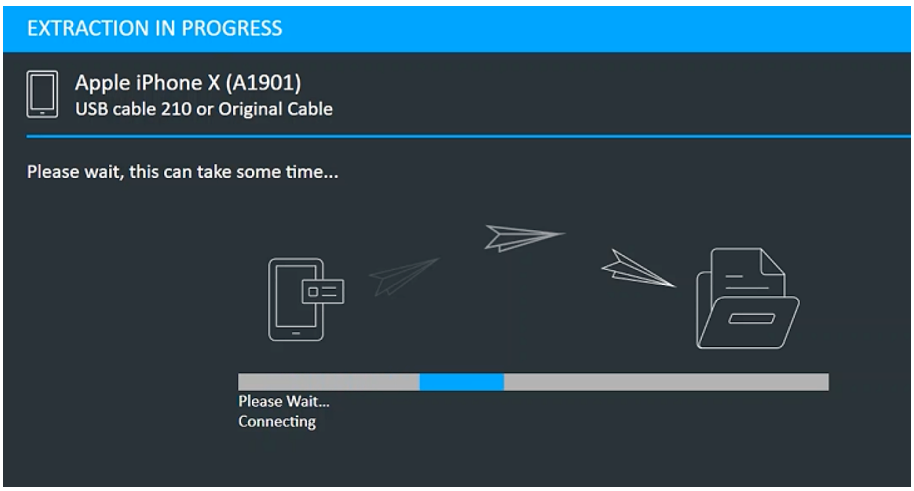


Figure 3.15: Full filesystem acquisition in progress using Cellebrite UFED

7. Once the device has been successfully exploited, the acquisition stage will start and the full filesystem will be imaged and archived in a DAR file, which is similar to a TAR archive

When the acquisition process ends, the device will automatically reboot. The resulting DAR archive can be analyzed by importing it into Cellebrite Physical Analyzer or the tool of your choice.

So far, we have focused on logical and filesystem acquisition techniques, which are the primary methods used during modern iOS investigations. For completeness, it is also important to understand physical acquisition, what it involves, why it was once considered the gold standard in mobile forensics, and why it is now largely unavailable on modern iOS devices.

iOS physical acquisition

Physical acquisition means making a bit-by-bit copy of a device's nonvolatile flash memory. In principle, a physical image is the most forensic-rich artifact you can obtain. It contains active files, system data, slack space, unallocated blocks, filesystem metadata, and any remnants that deletion left behind. This makes physical images extremely valuable for carving, recovering deleted records, examining APFS snapshots, and performing low-level timeline and corruption analysis. As of 2026, no publicly available technique exists for performing true physical acquisition on current-generation iPhones running iOS 26 while preserving access to encrypted user data.

Unfortunately, for modern iPhones and iPads, obtaining a useful physical image is not feasible. Apple designed iOS devices to protect user data with hardware-backed encryption, and the SEP plays a central role in that protection. The practical result for examiners is that a raw dump of flash memory is usually ciphertext without the ability to decrypt it.

Modern iOS devices use hardware-tied keys fused into the system-on-chip. Many user-level encryption keys are derived from those hardware keys combined with the user passcode. The Secure Enclave stores and enforces access to critical key material and prevents extraction of that keying material by design. The following actions happen as a result:

- A bit-for-bit copy of flash from most modern iPhones will yield encrypted data that cannot be interpreted without access to the appropriate keys
- Secure boot, Secure Enclave protections, and other kernel/hardware mitigations make it infeasible to extract those keys from the device on most iPhones without manufacturer cooperation or a validated privileged exploit
- The simple equation "physical image equals readable evidence" does not hold for most iOS devices released since the iPhone 5s generation

> **Note**
>
> As of this writing, the last iPhone model that still permits true physical acquisition is the iPhone 5c (introduced in 2013).

It's worth noting that Boot ROM-level exploits, including checkm8 (A5–A11) and the more recently disclosed usbliter8 (A12/A13), provide code execution ahead of iOS loading, but neither compromises the Secure Enclave itself. On SEP-equipped devices (iPhone 5s and later), this means even a successfully exploited device still cannot decrypt user data through this pathway alone. The raw image remains ciphertext. True physical acquisition in the traditional sense therefore remains confined to pre-Secure-Enclave hardware.

iOS diagnostic and unified log acquisition

Although logical, filesystem acquisition remains the primary method of collecting data from iOS devices, you should not overlook another valuable source of evidence: **diagnostic and unified logs**. Modern versions of iOS continuously record system events, application activity, hardware status, networking operations, and security-related events through Apple's logging frameworks. These logs often provide valuable contextual information that complements traditional forensic artifacts and, in some investigations, may contain evidence that is unavailable through other acquisition methods.

Unlike user-generated data such as messages or photographs, system logs primarily record how the operating system and applications behave over time. They can help you reconstruct granular timelines, determine when applications were launched or terminated, identify network activity, analyze system crashes, verify device state changes, and correlate events across multiple applications and services.

Understanding unified logs and sysdiagnose

Prior to iOS 10, Apple relied primarily on the Apple System Log (ASL) framework for recording system events. Beginning with iOS 10, Apple introduced the **unified logging** system, a significantly more efficient and privacy-focused framework designed to consolidate logging across the operating system. Today, nearly every component of iOS, including the kernel, system services, and third-party applications, generates unified log entries.

The unified logging framework records events using distinct log levels: default, info, debug, error, and fault. Depending on system configuration and logging policies, some entries are stored only temporarily in memory, while others are stored on disk as compressed trace files (`.tracev3`) within `/var/db/diagnostics/`. Because storage is finite, older log entries are continuously overwritten as new events are generated.

A key distinction in log acquisition lies between a standard sysdiagnose archive and a full unified log collection:

- **sysdiagnose**: A targeted diagnostic archive generated on demand by iOS. It aggregates current system status metrics, active process lists, kernel logs, crash reports, and a high-density snapshot of Unified Logs covering roughly the previous 24 hours.
- **Full unified log archives**: The raw local trace files stored on the device ring buffers, which often retain up to 30 days of historical records depending on overall system activity.

Forensic value of unified logs

Although diagnostic logs do not replace full filesystem or physical acquisitions, they provide valuable supporting evidence that can significantly strengthen an investigation. Examples of forensic artifacts commonly recovered from these logs include these:

- Application launches, terminations, and background execution context
- Device boot, shutdown, restart, and sleep events
- Wi-Fi, Bluetooth, VPN, and network connectivity state changes
- USB connections, host trust states, and pairing activity
- System crashes, kernel panics, and application exceptions
- AirDrop, Continuity, and Handoff interaction telemetry
- Authentication events, passcode entries, and security-related operations
- Battery, charging, and thermal management metrics
- Diagnostic information useful for malware analysis and incident response

When combined with logical or filesystem acquisitions, these events help you build precise, second-by-second timelines and correlate user actions with underlying system activity.

Collecting diagnostic and unified logs

You can extract logs through native hardware triggers, operating system command-line utilities, or specialized commercial forensic suites. Let's discuss how to trigger sysdiagnose on the physical device.

If a device is unlocked and interactive, a sysdiagnose can be manually generated directly on the device using hardware button combinations:

- **Devices with a physical home button**: Press and hold both Volume buttons simultaneously with the home button for 1 to 1.5 seconds
- **Devices with face ID**: Press and hold both volume buttons simultaneously with the side (power) button for 1 to 1.5 seconds

> **Note**
>
> Note that the device registers a short haptic vibration or screen flash when triggered. The operating system then packages the diagnostic logs in the background, which typically takes 3 to 5 minutes to complete. Also, the trigger sequence has frequently changed across iOS versions and hardware generations, and hence, you should verify the appropriate procedure for the device under examination.

Manual extraction pathways (without FFS)

While a subsequent full filesystem extraction will capture stored sysdiagnose packages under `/var/mobile/Library/Logs/CrashReporter/DiagnosticLogs/`, they can also be retrieved manually when full filesystem access is unavailable:

- **Direct UI sharing**: Navigate to **Settings** | **Privacy & Security** | **Analytics & Improvements** | **Analytics Data**. Locate the target archive (prefixed with `sysdiagnose_` and a timestamp) and export it via AirDrop or save it to the local **Files** application.
- **Workstation sync directory**: Connecting the target device to a trusted analysis machine transfers generated crash reports and diagnostic packages during local sync operations. These are deposited at:
  - **macOS**: `~/Library/Logs/CrashReporter/MobileDevice/<Device_Name>/`
  - **Windows**: `%AppData%\Roaming\Apple Computer\Logs\CrashReporter\MobileDevice\<Device_Name>\`

Acquiring full unified log archives (~30 days)

To capture the full 30-day historical trace beyond the 1-day snapshot included in a sysdiagnose, native command-line tools or specialized utilities must be used:

- **macOS command-line utility**: On a paired, trusted macOS workstation, you can acquire the complete Unified Log archive from a connected iOS device using the log collect command in Terminal:

```
Bash
log collect --device --output logs.logarchive
```

- **Commercial forensic suites**: Solutions such as Elcomsoft iOS Forensic Toolkit (EIFT), Magnet AXIOM, Cellebrite UFED, and Oxygen Forensic Detective provide dedicated

workflows to trigger and pull diagnostic archives and full unified log records via developer protocols.

The resulting `.logarchive` bundle can be opened natively in Apple's Console application on macOS or imported into specialized forensic parsing tools (such as `mac_apt` or custom Python scripts) for structured timeline queries.

Limitations and best practices

Like any forensic artifact, diagnostic logs have limitations. Unified Logs are designed primarily for real-time system diagnostics rather than long-term evidence preservation. Their retention period depends on available storage, device activity, and internal logging policies, meaning critical historical events may be overwritten quickly on high-usage devices. Additionally, certain log entries are intentionally redacted by the operating system (labeled as `<private>`) to protect user privacy, and the sheer volume of generated events requires rigorous filtering.

Despite these limitations, diagnostic, and unified logs remain a vital complementary acquisition vector. Whenever supported by the device and investigative scenario, you should collect logs as early as possible during an examination, preserve cryptographic hashes of acquired `.logarchive` bundles, and document the chain of custody alongside traditional logical, file system, and cloud evidence.

Decision framework for selecting iOS acquisition methods

Not every iOS device can be acquired using the same method, and not every method yields the same results. The right approach depends on a combination of factors: the device model, the processor generation, the version of iOS installed, whether the device is unlocked, and the legal authority granted to the examiner. This section provides a structured way to decide which acquisition method is most appropriate for a given scenario.

Step 1: Starting with the device model and processor generation

The very first step is to identify the exact iPhone or iPad model. Processor generation is crucial because it determines what exploits, if any, are available:

- **A5–A11 devices (iPhone 5s through iPhone X)**: These are vulnerable to the hardware-based checkm8 exploit, which enables jailbreak-assisted filesystem acquisition

regardless of iOS version. This makes them the most examiner-friendly devices still in circulation.

- **A12 and newer devices (iPhone XS/XR, iPhone 11 and beyond)**: Practical acquisition options remain limited. Although some recent Boot ROM research has targeted portions of this device family, it has not fundamentally changed the forensic acquisition workflow for modern iPhones. Investigators generally rely on logical acquisition, agent-based methods (where supported), cloud acquisition, or commercial forensic tools.

Step 2: Considering the iOS version

iOS updates frequently patch software vulnerabilities that forensic tools may rely upon:

- On older iOS versions, public jailbreaks may exist that enable full filesystem acquisition.
- On the latest releases (iOS 26 and beyond), public jailbreaks are rare or non-existent for modern hardware. Commercial forensic vendors sometimes maintain private exploits, but these are not publicly documented.

In practice, the newer the iOS version, the more likely it is that acquisition options will be limited to logical and cloud-based methods.

Step 3: Locked or unlocked state of the device

The device's state at seizure is often decisive:

- **Unlocked device**: If the device is seized while unlocked, or if a trusted pairing record is available, logical or filesystem acquisition may be possible.
- **Locked device**: If the device is locked and the passcode is not known, options shrink dramatically. On modern devices with the Secure Enclave, brute-force attacks are generally infeasible. You may only be able to preserve the device in its current state until credentials are obtained through legal or investigative means.

Step 4: Matching legal authority with acquisition depth

As a forensic examiner, you must always align the acquisition method with the legal scope of the warrant or consent. Some methods, such as jailbreak-assisted acquisition, alter the device and may raise questions in court. It is essential to confirm that the authority extends to methods that involve modification of the device.

Step 5: Mapping acquisition options

Based on the factors previously mentioned, the decision tree typically looks like this:

- **Legacy devices (pre-iPhone 5s)**: Physical acquisition may still be possible.

- **iPhone 5s–iPhone X (A7–A11)**: Checkm8-based acquisition may enable filesystem extraction on supported devices. However, the exact acquisition capabilities depend on the device model, iOS version, and security state (such as passcode configuration), so you should always verify current tool support and prerequisites before attempting acquisition.
- **iPhone XS and newer (A12/A13)**: As of June 2026, the usbliter8 Boot ROM exploit may enable limited low-level access on this subset, but it does not bypass the Secure Enclave, so it does not currently change the practical acquisition path. Continue to rely on logical backup, agent-based acquisition (if unlocked), or cloud acquisition. Commercial tools such as Cellebrite UFED, Magnet Graykey, or Oxygen Forensics remain the primary options where available.
- **Locked modern devices without passcode or pairing record**: Realistically, acquisition is limited. Only cloud data or partial artifacts (such as from synced computers) may be available.

Step 6: Balancing completeness and risk

Each acquisition method comes with trade-offs:

- **Logical acquisition**: Safest, least invasive, but limited in scope
- **Filesystem acquisition**: Rich in data but may require exploits or modifications
- **Physical acquisition**: Only feasible on very old models, but the richest form of evidence when available

You must weigh the depth of evidence needed against the feasibility and risks of the chosen method. Careful documentation of why a specific method was selected is critical for both technical soundness and legal defensibility.

Summary

Acquiring data from iOS devices remains one of the most complex yet crucial aspects of mobile forensics. Unlike traditional computers, iPhones and iPads are built with layered hardware and software security that directly impacts what data can be extracted and how. In this chapter, we examined the different operating modes of iOS devices, such as normal, recovery, and DFU, and why they matter in forensic workflows. We also discussed the unique legal and forensic considerations surrounding iOS, from the role of passcodes and the Secure Enclave to the risks posed by remote wipes and iCloud synchronization.

We explored the range of acquisition methods available, beginning with logical extraction and moving into more advanced approaches like agent-based and jailbreak-assisted filesystem acquisitions. While logical acquisition remains the most accessible, filesystem acquisition

provides far deeper access to application data, logs, and system artifacts. We also addressed physical acquisition, clarifying that it is only possible on very old iPhone models such as the iPhone 5c and earlier, with modern devices offering no viable path due to encryption and Secure Enclave protections.

Finally, the chapter presented a decision framework to guide you in selecting the appropriate acquisition method based on device model, processor generation, iOS version, device state, and legal authority. The chapter emphasized that documentation, validation, and adherence to legal scope are just as critical as the technical process itself. The next chapter will build on this foundation by focusing on data acquisition from iOS backups, an approach that can often provide valuable artifacts even when full device access is restricted.

4

Data Acquisition from iOS Backups

In the previous chapter, we explored methods for acquiring data directly from iOS devices, including logical and filesystem extractions. While those approaches can provide the richest set of artifacts, they are not always possible, particularly with modern iPhones and iPads where encryption and hardware protections may block deeper access. In many investigations, the most practical and sometimes the only source of data comes from **iOS backups**.

Backups occupy a unique position in mobile forensics. They are designed by Apple as a user-friendly way to safeguard device data. Yet, for examiners, they provide an indirect but highly valuable window into a suspect's digital activity. Each time an iPhone or iPad syncs, the system may generate a backup either on the user's computer or on iCloud. These backups can contain communications, photos, browsing history, application data, and even sensitive records such as Wi-Fi keys and health information, depending on whether the backup is encrypted.

Backups are especially important because they often persist beyond the lifetime of a device. A phone may be lost, destroyed, or locked with a passcode that resists bypass, but if a backup exists, much of the same data may still be accessible. This makes backups not only a supplement but sometimes the central evidence source in an investigation.

In this chapter, we will cover the following topics:

- Introduction to iOS backups
- Local backups using Finder, Apple Devices and iTunes
- iCloud backups

By the end of this chapter, you will understand how backups fit into the broader forensic toolkit, how to approach them depending on the device and iOS version, and how to maximize the evidentiary value of backup data while staying mindful of its limitations.

Introduction to iOS backups

When investigators are unable to perform full physical or filesystem extraction of an iOS device, backups often become the most practical and valuable alternative. An iOS backup is a structured collection of data selected by Apple's backup service for restoration. It is not a complete image or exact point-in-time snapshot of the device. Its contents depend on the iOS version, application settings, backup type, encryption state, and whether particular data is synchronized separately through iCloud. Backups are a valuable repository of artifacts. Backups are not merely secondary options when more direct acquisitions fail. In fact, they often serve as the primary evidence source in modern investigations.

As discussed in earlier chapters, Apple's Secure Enclave, hardware-backed key management, secure boot chain, and file-level data protection make low-level acquisition difficult on modern devices. The available acquisition methods depend on the device model, processor generation, iOS version, security patch level, lock state, and tool support. Local backups may provide an alternative when the device is unlocked or a valid pairing relationship exists, while existing computer or iCloud backups may remain valuable when direct device acquisition is unavailable. Access still depends on factors such as legal authority, backup passwords, Apple account authentication, two-factor authentication, trusted-device status, and **advanced data protection** (**ADP**) settings.

In this section, we will explore why backups play such an important role in forensic workflows, how they differ from other acquisition types, and the two primary categories of backups available to examiners: local backups and iCloud backups.

The role of backups in forensic investigations

Most iOS users rely on backups, often without realizing it. Apple's ecosystem is designed to quietly preserve device data, whether through automatic iCloud backups or periodic synchronization with a computer. By default, Apple encourages users to back up their data either locally (to a computer using iTunes or Finder) or to iCloud. These backups include a wide array of personal data such as messages, photos, app data, browsing history, Wi-Fi settings, health information, and more.

For forensic analysts, iOS backups often provide the following information:

- **Historical backup states**: A device may have been reset, upgraded, or even destroyed, yet its earlier backups may still exist on a suspect's computer or in their iCloud account. This allows you to recover artifacts from periods before the device was seized.

- **Broader context**: iCloud backups can contain synchronized data from multiple devices using the same Apple ID, revealing patterns of activity that go beyond the seized handset.
- **Accessible evidence**: Unlike physical images, backups are stored in user-space formats (databases, plist files, media directories), making them easier to parse with forensic tools and interpret in court.

Backups can therefore fill crucial gaps when live device access is impossible. For example, if a suspect enables the "Erase Data after 10 Failed Passcode Attempts" feature and refuses to provide the passcode, as the forensic examiner, you can still analyze an existing iCloud backup tied to the same Apple ID.

Backups vs physical and logical acquisition

It is important to understand where backups fit in the hierarchy of forensic acquisition methods:

- **Physical acquisition** traditionally refers to obtaining a bit-for-bit or low-level representation of device storage. On supported legacy devices, it may expose filesystem structures, unallocated space, and residual data. However, on modern encrypted iPhones, a raw storage image may contain less usable evidence than a decrypted filesystem or backup acquisition.
- **Filesystem acquisition** provides access to some or all the accessible directory structure, potentially including application containers, databases, logs, caches, and system artifacts. This method is largely possible only when the device is unlocked or when an exploit (such as checkm8 on older hardware) is available.
- **Logical acquisition** uses operating-system services and supported protocols to collect selected user-accessible data. It generally provides less filesystem context than filesystem acquisition, although the precise contents vary by acquisition method, iOS version, and tool. Deleted records, temporary files, caches, and many system-level artifacts are commonly absent.

Backups sit between logical and filesystem acquisition. They are neither a complete representation of the device nor a simple subset. Instead, they represent a curated snapshot created by Apple's backup service. Backups generally include the structured content that Apple considers important for device restoration such as messages, photos, contacts, key application databases, Safari data, Keychain items (if encrypted), and numerous configuration files. While backups do not provide a raw representation of deleted data, residual information may still exist within SQLite databases, application files, caches, and other artifacts included in the backup.

The most important distinction is **availability**. While filesystem extraction may be blocked by encryption, secure boot protections, or the lack of a jailbreak, backups remain accessible in many situations where no other acquisition method is possible. A device may be locked, damaged, destroyed, or remotely wiped, but its historical backups may still exist on a suspect's computer or reside in their iCloud account. In such cases, backups become not just an alternative method but the primary and sometimes only source of actionable forensic evidence. For these reasons, understanding the structure, limitations, and forensic potential of both local and iCloud backups is essential for modern iOS investigations.

Local vs. iCloud backups

Apple provides users with two main backup options, and the choice between them has direct implications for forensic analysis:

- **Local backups** (iTunes and Finder): When a user connects an iPhone or iPad to a computer, the device can be backed up locally. Finder manages device backups on macOS Catalina and later. On current Windows systems, Apple recommends the Apple Devices app; iTunes may still be encountered on older systems or computers on which Apple Devices is not installed.
  Local backups are stored as a structured directory containing files identified by hashed file identifiers rather than their original filenames. Metadata such as `Manifest.db`, `Manifest.plist`, and `Info.plist` provide the information needed to reconstruct and interpret the backup contents. Local backups may be unencrypted (easily parsed) or encrypted (which unlocks additional sensitive data such as saved passwords, health data, and Wi-Fi keys).
- **iCloud backups**: With iCloud, Apple introduced the option to back up directly to cloud storage. iCloud backups are tied to the user's Apple ID and typically occur automatically when the device is charging and connected to Wi-Fi. These backups often include much of the same data as local backups, but the scope can differ due to storage limitations or user settings.
  Accessing iCloud backups requires Apple ID credentials, an authentication token, or cooperation from Apple via legal request. Since the rollout of ADP in 2023, many categories of iCloud data are now end-to-end encrypted, making examiner access even more challenging without the user's keys.

Table 4.1 presents a detailed comparison of local vs. iCloud backups:

Feature / attribute	**Local backups (iTunes / Finder)**	**iCloud backups**
Storage location	Stored on user's computer under the `MobileSync` directory	Stored remotely in Apple's iCloud infrastructure
How are they created	Manual or automatic when the device is connected to a trusted computer	Automatic when the device is charging, locked, and connected to Wi-Fi
Backup format	Directory of hashed files with metadata (`Manifest.db`, `plist files`)	Apple-managed encrypted archive stored on cloud servers
Encryption	Optional; encrypted backups unlock more categories (Keychain, Health, Wi-Fi)	Encrypted by default; many categories are end-to-end encrypted under ADP
Data available	Typically broad: messages, photos, app data, Keychain (if encrypted)	Substantial, but may vary with storage limits and ADP settings
Dependency on device state	Requires device to be unlocked or previously paired	Requires Apple ID + password + MFA or a valid authentication token
Access requirements	File system access to backup directory; can be parsed with forensic tools	Apple ID credentials, token-based authentication, or Apple's legal compliance process
Risk factors	User may delete old backups; computer may be unavailable	Data availability depends on user storage plan and ADP configuration

Feature / attribute	Local backups (iTunes / Finder)	iCloud backups
Best use cases	Historical or large backups; deep analysis of structured artifacts	Recent snapshots; devices that are locked, wiped, or destroyed
Impact of advanced data protection (ADP)	No impact; local backups behave the same	Major impact; many categories become inaccessible without user keys
Forensic value	High, especially for encrypted backups	High, but dependent on cloud sync settings and encryption policies

Table 4.1: Comparison of local vs. cloud backups

In practice, you should consider both local and iCloud backups when they fall within the scope and legal authority of the investigation. A computer at a crime scene may contain months of local backups in hidden directories, while iCloud may hold a more recent snapshot of the same device. Combining both sources often yields the most complete evidence set. We will now examine local backups, including how they are created, located, and analyzed, before exploring iCloud backups later in the chapter.

Local backups using Finder, Apple Devices, and iTunes

Before connecting an evidentiary device to a computer, determine whether an existing backup from that device is already stored on the system. Preserve the existing backup before initiating a new one. Creating a new backup may update or replace backup content associated with the device and may alter the historical evidence available on the computer.

You should also record the device's power state, lock state, BFU or AFU state, pairing status, battery level, and existing backup-encryption configuration before beginning. Create a new backup only when it is authorized, necessary, and supported by a validated acquisition procedure.

Local backups can be created and examined using several methods depending on the operating system and tools available. The following sections explain how to create a local backup and identify the files and metadata it contains.

Creating local backups

Local backups are one of the most common sources of forensic evidence from iOS devices. A local backup may be created through **Finder** on current macOS systems, **Apple Devices** on current Windows systems, or **iTunes** on older supported systems. The device normally must be unlocked to establish a new trust relationship. A previously paired computer may retain a valid pairing relationship, although its usefulness depends on the device state, pairing records, iOS version, and authentication requirements.

When an iPhone or iPad is connected to a Mac via USB, Finder automatically detects the device under the **Locations** sidebar. As shown in *Figure 4.1*, you can select **Back up all the data on your iPhone to this Mac** and initiate the process by clicking **Back Up Now**:

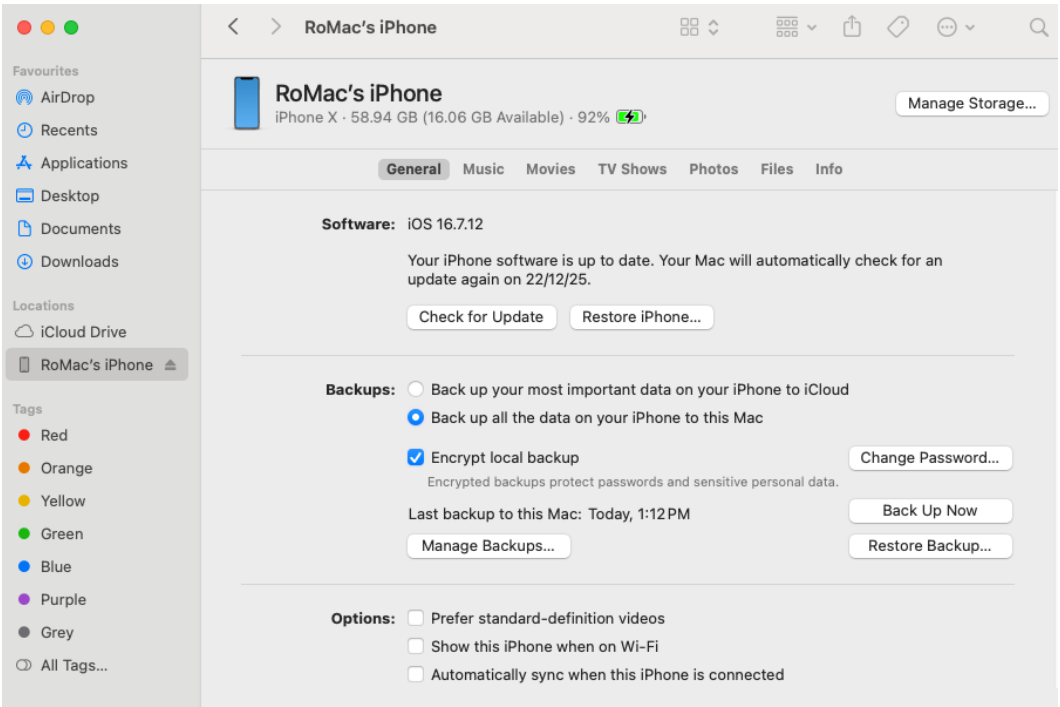


Figure 4.1: iPhone connected with local backups enabled

If encryption is enabled, Finder will prompt you to enter a password to protect sensitive data categories. Later, you must enter this password every time you need to access the encrypted backups.

Locating local backups

As shown in the following figure, on macOS, backups are stored in `/Users/<username>/Library/Application Support/MobileSync/Backup/`:

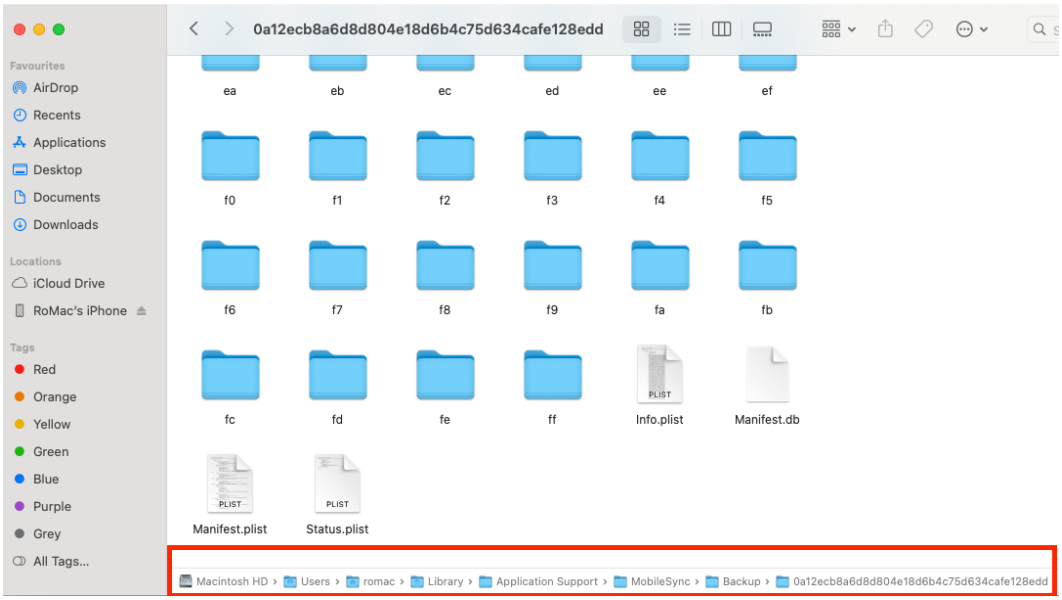


Figure 4.2: iPhone backup folder on a macOS computer

On Windows, the backup location depends on the version of iTunes being used. The standalone version of iTunes typically stores backups under `Users\<username>\AppData\Roaming\Apple Computer\MobileSync\Backup\`.

If iTunes was installed from the Microsoft Store, backups are typically stored under `\Users\<username>\Apple\MobileSync\Backup`.

Each device backup is stored in a uniquely named directory associated with the backed-up device. The directory name and identifier format can vary across device generations and software versions. Inside the directory, files are stored under hashed identifiers rather than their original paths and filenames. You should confirm attribution using metadata such as `Info.plist` rather than relying solely on the backup-directory name.

Each backup process generates metadata files (`Info.plist`, `Manifest.db`, etc.) that describe the backup's contents, and you can later confirm backup timestamps, device identifiers, and encryption status from these files.

Local backup behavior depends on the host application, device settings, and user actions. Do not assume that every synchronization created a complete backup. Determine the backup date, status, device attribution, and completeness from the available metadata and acquisition context. A computer may contain backups associated with multiple devices or historical backup states. The directory structure alone does not establish that each folder is a complete, independent point-in-time backup. For example, a message that appears in an older backup but is missing from a newer one may indicate that it was deleted.

Standard backup metadata files

Alongside user data, Apple includes several standard files that describe the device and the backup. Since iOS 10, the following files are always present:

- **Info.plist**: Contains high-level device information such as device name, serial number, IMEI, phone number, product type, iOS version, and list of installed applications. This file is often the examiner's first checkpoint for linking a backup to a physical device.
- **Manifest.plist**: Describes backup contents and includes details such as backup version, creation date, whether the backup is encrypted, whether a passcode was set on the device, and the "backup keybag" used to re-encrypt data for secure storage.
- **Status.plist**: Stores information about the backup's state, such as whether it was a full or incremental backup, the last modification timestamp, and whether the backup process completed successfully.
- **Manifest.db**: An SQLite database that serves as the core index for the backup. It maps hashed file identifiers in the backup to their original file paths and domains on the device (for example, messages, photos, application data). The database also contains file metadata such as creation and modification timestamps.

Prior to iOS 10, backups used a simpler structure with `manifest.mbdb`, but this has now been fully replaced by the SQLite-based `manifest.db`.

Backup contents

The data included in a local backup depends on the iOS version and whether backup encryption is enabled. In recent iOS versions, some artifact categories are available only in **encrypted backups**. Encrypted backups generally contain a broader set of data, including sensitive artifacts that are omitted from unencrypted backups. This makes encrypted backups particularly valuable in forensic analysis, though access requires knowledge of the backup password.

Since iOS 10, Apple also reorganized backup storage into a directory structure of **two-character hexadecimal folders**. These folders contain the hashed file segments that correspond to the original files on the device, as mapped in `manifest.db`. This structure

optimizes lookup and parsing but makes manual browsing impractical without a database viewer or forensic tool.

Encrypted vs. unencrypted backups

During investigation, you may encounter situations where the examined backup is encrypted. This happens when the owner of the device used an encryption option when they created backups. The following is the key difference between encrypted and unencrypted backups:

- **Unencrypted backups**: These contain a limited subset of device and application data. In recent iOS versions, several artifact categories that were previously available in unencrypted backups may only be included when backup encryption is enabled.
- **Encrypted backups**: Encrypted computer backups may contain data categories omitted from unencrypted backups, including saved passwords, Wi-Fi settings, website history, health data, and call history. Access to the protected contents requires the backup password. Encrypted backups do not contain face ID and touch ID templates or the device passcode.

> **Note**
>
> Always determine whether local backup encryption is enabled before beginning acquisition. Encrypted backups can contain significantly more forensic data than unencrypted backups, including additional Keychain, Health, and Wi-Fi-related artifacts. However, the backup password is required to decrypt and analyze the protected content. Document the encryption state and preserve any available password or credential information as part of the acquisition process.

Creating a local iOS backup is one of the simplest and safest ways to extract data from a device, especially when file system or physical acquisitions are not possible. Since modern iPhones use hardware-backed encryption and the Secure Enclave to restrict low-level access, backups often become the primary source of evidence available to investigators.

A local backup is typically generated using Finder on modern macOS systems, Apple Devices on current Windows systems, or iTunes on older systems where it is still in use. These applications use Apple's built-in synchronization service to communicate with the device over USB or Wi-Fi, creating a structured copy of user data on the host computer.

Extracting backups on macOS

Once a local backup has been identified or created, preserve it in place where practicable, acquire or copy it using the laboratory's validated procedure, and analyze a verified working copy on the forensic workstation. Direct interaction with iTunes or Finder should be avoided

after seizure to prevent overwriting or syncing changes. Always create a read-only copy or forensic image of the backup directory before beginning analysis.

Forensic examiners typically use specialized tools to parse backup data into a readable format. Popular solutions include the following:

- **Cellebrite UFED and Physical Analyzer**: Automatically recognizes and parses iTunes and Finder backups, extracting structured artifacts like messages, calls, Safari history, and app data
- **Oxygen Forensic Detective**: Provides detailed reporting and visualization of backup data, including encrypted backups (if the password is known)
- **Elcomsoft Phone Breaker**: Supports decryption of local and iCloud backups, as well as password recovery for encrypted ones
- **libimobiledevice (open-source)**: Offers command-line utilities such as `idevicebackup2` to create and extract backups manually
- **iBackup Viewer and iExplorer**: Lightweight tools useful for quickly reviewing backup contents or exporting specific data categories

Each of these tools reads the `Manifest.db` database to reconstruct the original directory tree, filenames, and metadata. Once parsed, artifacts such as SMS, call history, and app-specific data can be exported for deeper analysis, correlation, and reporting.

Forensic best practices

When creating or collecting backups for forensic use, follow these best practices:

- **Ensure the device is trusted**: The computer must have a valid pairing record; otherwise, the backup cannot be initiated
- **Prefer encrypted backups**: They contain more comprehensive data, but the password must be documented or recovered
- **Preserve integrity**: Hash the backup folder immediately after acquisition and maintain a chain of custody
- **Avoid automatic syncs**: Disable Wi-Fi and automatic device syncing in iTunes/Finder to prevent inadvertent data changes
- **Document timestamps**: Record backup start and completion times, as well as file modification times, to establish a clear timeline

By following these steps, you can extract and preserve iOS backups in a manner that ensures both forensic soundness and evidentiary value. The following practical walkthrough demonstrates how an iOS backup can be created and extracted using the open-source `libimobiledevice` toolkit.

Practical walkthrough: Extracting an iOS backup using idevicebackup2 (open-source)

The open-source `libimobiledevice` suite allows you to create and extract backups manually without using iTunes or Finder. It communicates with iOS devices using Apple's native protocols and is widely used.

On the mac workstation, install **homebrew**, which is a package manager, and then install the `libimobiledevice` package. Connect the iPhone to the mac workstation and follow these steps:

1. Open a terminal and run `ideviceinfo`. If the pairing record is valid, the device's hardware and software identifiers will be displayed:

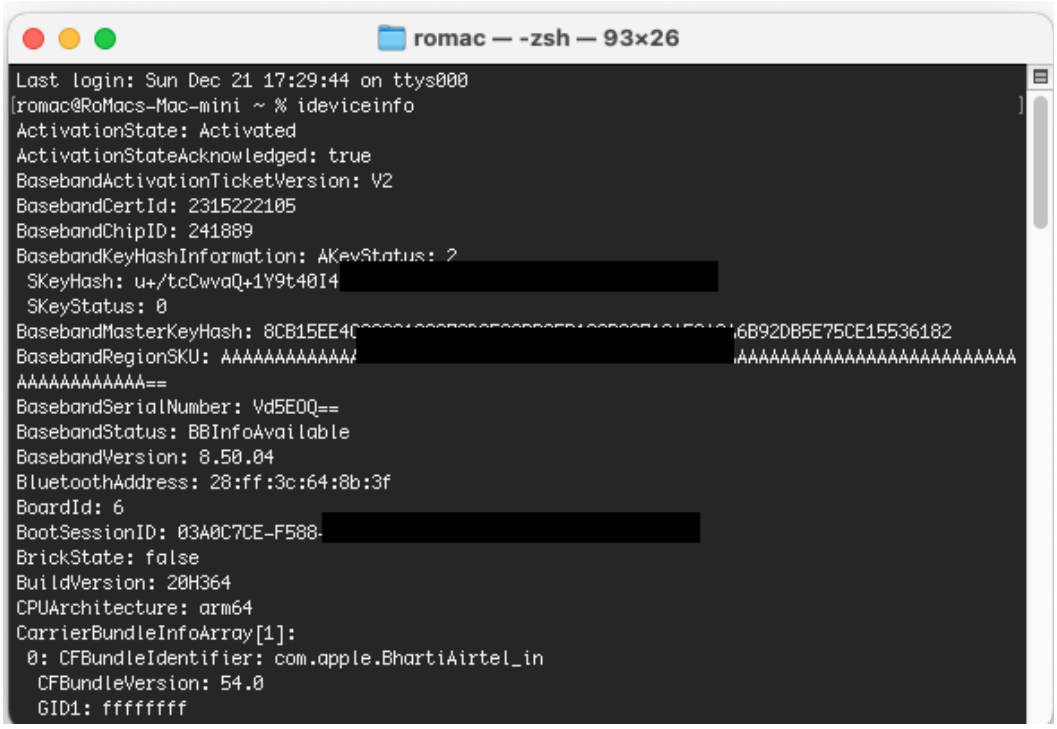


Figure 4.3: Output of ideviceinfo command for a device successfully connected

   If pairing is not established, run `idevicepair pair`. You must unlock the device and tap **Trust This Computer**.

2. Now run the command to create a backup directory `mkdir ios_backup`

3. To start backup extraction, run the command `idevicebackup2 backup ios_backup`:

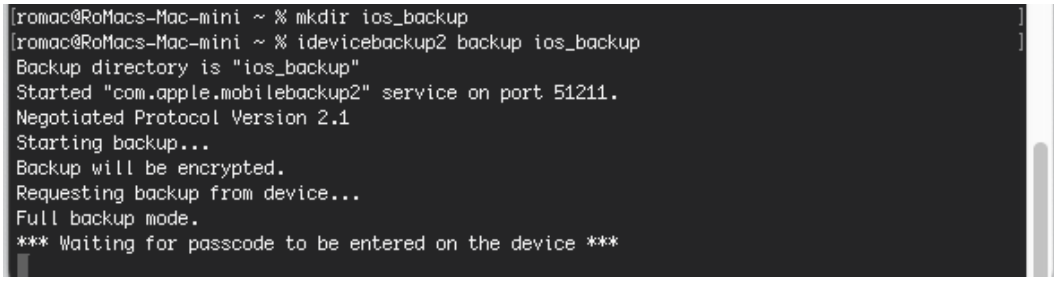


Figure 4.4: Creating an encrypted iOS backup using idevicebackup2

If backup encryption is already enabled, the backup created by the device will remain encrypted. Depending on the operation and the version of idevicebackup2, you may be asked to enter the device passcode on the device or provide the backup password interactively. A completed encrypted backup may still be preserved without decrypting it, but its protected contents cannot be meaningfully parsed until the correct backup password is supplied.

4. The tool will now begin copying the backup files (`Manifest.plist`, `Info.plist`, `Manifest.db`, and the file domain folders) into the specified directory:



Figure 4.5: idevicebackup2 copying backup files from the iOS device to the forensic workstation

5. Once the backup is complete, you will see a **Backup Successful** confirmation message as shown here:

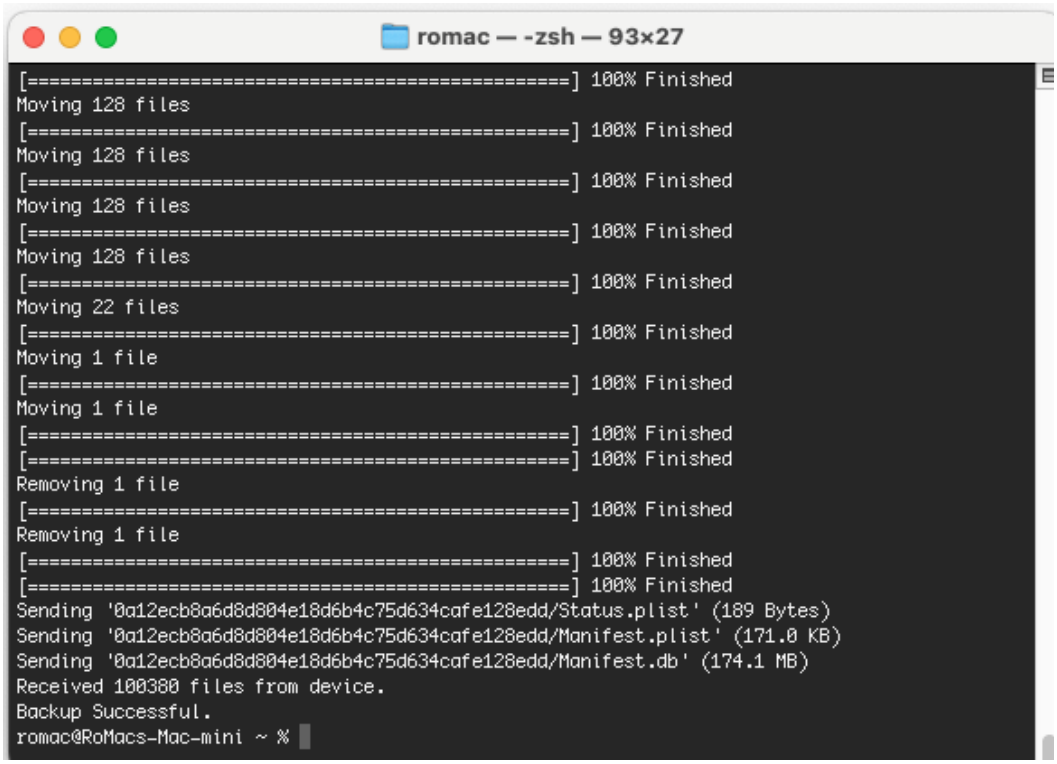


Figure 4.6: Successful completion of an iOS backup using idevicebackup2

It is recommended to create a sorted `SHA-256` manifest containing the hash of every file in the backup: `find ios_backup -type f -exec shasum -a 256 {} \; | LC_ALL=C sort > ios_backup.sha256`.

Preserve the resulting manifest alongside the backup and record the acquisition tool, tool version, source device, start, and completion times, destination path, and any acquisition errors. If the laboratory uses a forensic container or an archive created according to a validated procedure, also calculate, and record the SHA-256 hash of that container or archive.

Tip

Preserve the acquired backup before beginning analysis. Create a working copy and calculate cryptographic hashes of the acquired files or forensic containers, where supported by your workflow. Record the hash values, acquisition time, tool, and version

> used, and backup location in case notes. Perform subsequent analysis on the working copy rather than the preserved original.

This process produces a computer-backup acquisition using Apple's backup service. It is not a complete representation of the device and should not be equated with a filesystem or physical acquisition. Its forensic reliability depends on validated tooling, complete preservation of the output, cryptographic verification, contemporaneous documentation, and a clear account of any changes introduced on the device. The directory can now be parsed for artifacts such as SMS, call logs, app data, Safari history, and third-party application databases. You can now open the backup using tools such as iBackup Viewer, Cellebrite Physical Analyzer, or Oxygen Forensic Detective.

Once a backup has been successfully acquired and preserved, the next step is to examine its contents and identify relevant artifacts. The following walkthrough demonstrates how an iOS backup can be reviewed using iBackup Viewer.

Practical walkthrough: Analyzing an iOS backup using iBackup Viewer

iBackup Viewer is a lightweight macOS and Windows tool that allows you to browse the contents of an iTunes/Finder backup without modifying it. While it does not provide advanced parsing like commercial forensic suites, it is extremely useful for quickly verifying backup integrity, reviewing file structures, and exporting key artifacts.

We will now look at a step-by-step process to use iBackup Viewer to analyze the backup file created in the earlier section.

1. Visit the official website`https://www.imactools.com/iphonebackupviewer/` and download the macOS or Windows version.
2. Install the application and launch **iBackup Viewer**.
3. When opened, the tool automatically scans the following default backup directory:
    - **macOS**: `~/Library/Application Support/MobileSync/Backup/`
    - **Windows (standalone iTunes):**
    - `C:\Users\<username>\AppData\Roaming\Apple Computer\MobileSync\Backup\`
    - **Windows (Microsoft Store iTunes):**
    - `C:\Users\<username>\Apple\MobileSync\Backup\`

Place the working copy in a supported backup location or configure iBackup Viewer to use the directory containing the working copy. Do not direct the tool to the preserved evidentiary copy unless its read-only behavior has been validated.

> **Note**
>
> On macOS, iBackup Viewer may require **Full Disk Access** to read backups stored under protected system locations. If an existing backup is not detected or cannot be opened, check **Privacy & Security | Full Disk Access** and grant access to iBackup Viewer if required. Only grant this permission on the forensic workstation being used for the examination and remove it afterward if it is no longer required.

4. Each entry displays identifying information about the available backup. In our case, since the backup is encrypted, **iBackup Viewer** in the following figure can be seen prompting for the backup password before allowing access:

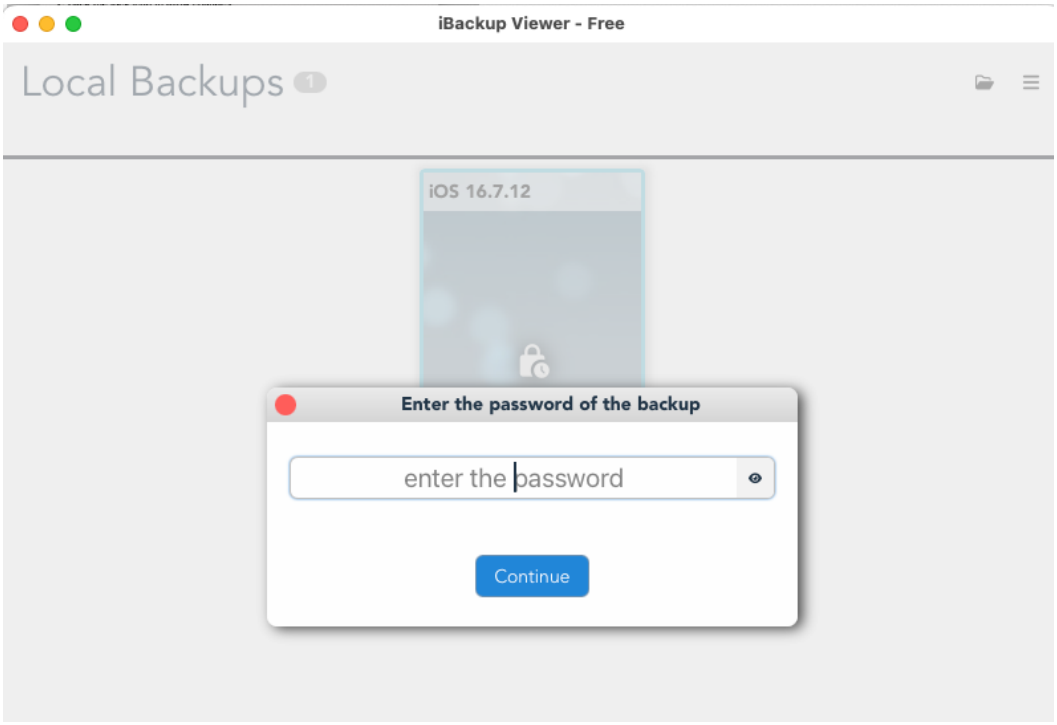


Figure 4.7: iBackup Viewer prompting for the password of the encrypted iOS backup

5. Once the backup opens, you will see a set of organized categories such as the following:



Figure 4.8: iBackup Viewer displaying organized application and data categories from the opened iOS backup

These categories are parsed using the backup metadata (`Manifest.db` and associated `plist files`).

6. Clicking on any individual category will show you specific details:

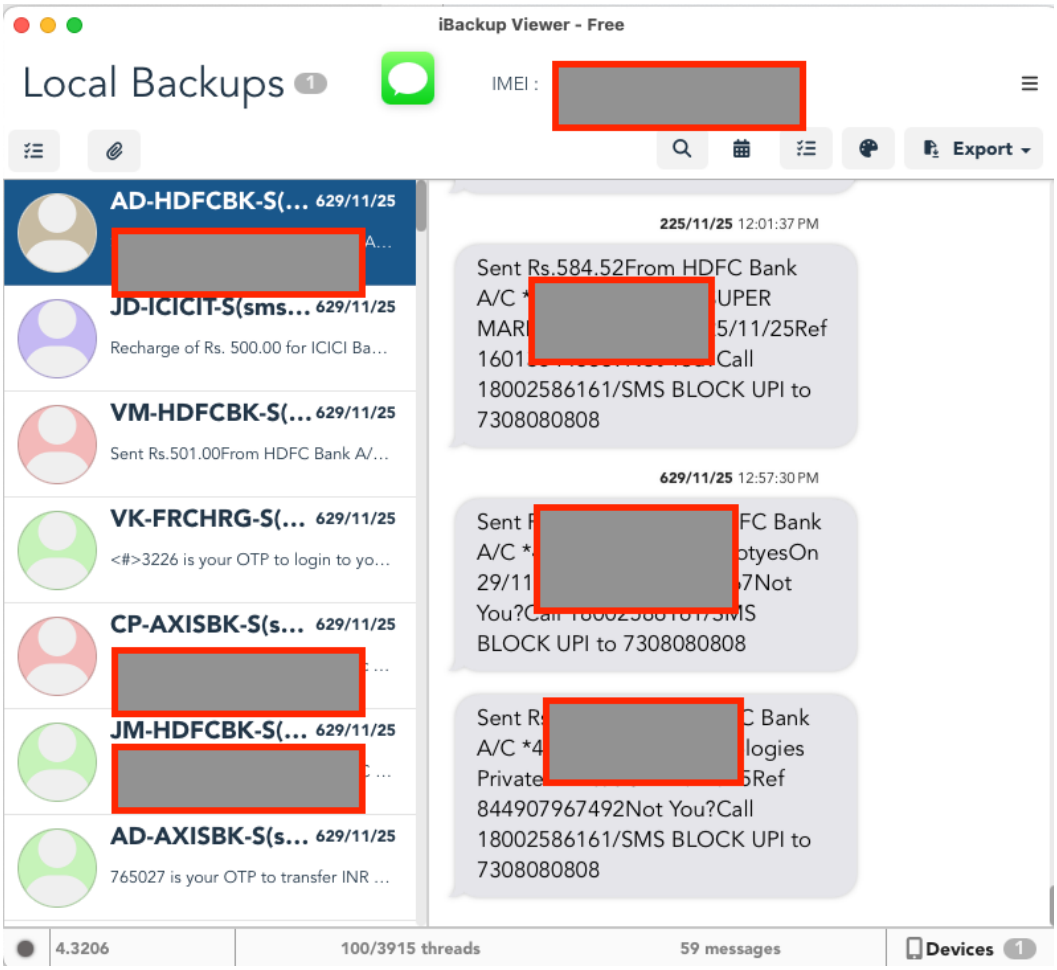


Figure 4.9: SMS messages extracted from an iOS backup using iBackup Viewer

In addition to messages, the tool can also be used to review other artifact categories. For example, *Figure 4.10* shows photographs recovered from the device's camera roll, allowing you to browse media contained within the backup:

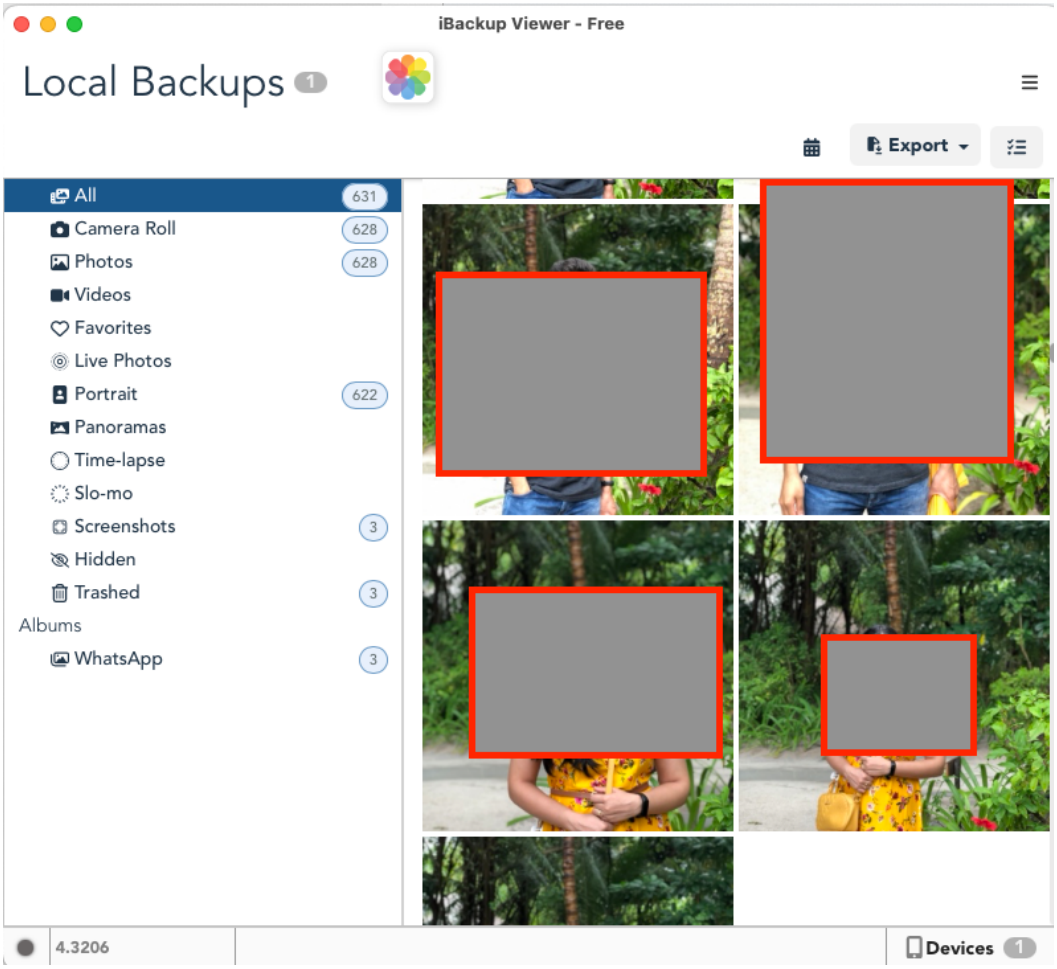


Figure 4.10: Photo thumbnails from the Camera Roll extracted from an iOS backup using iBackup Viewer

While local backups provide a valuable source of forensic evidence, you should also consider data stored in iCloud, which may provide additional or more recent information.

iCloud backups

iCloud backups have become a significant source of forensic evidence, particularly as Apple continues to strengthen on-device encryption and restrict access to full file system extractions. When direct device acquisition is restricted or the device is unavailable, an existing iCloud backup may provide an alternative source of user data. Its availability and contents depend on the account configuration, backup state, authentication material, legal authority, and whether

ADP is enabled. Understanding how these backups work, what data they contain, and how to lawfully access them is essential for modern mobile forensic investigations.

Understanding how iCloud backups work

When iCloud backup is enabled, iOS periodically backs up eligible device data while the device is locked and connected to power and a suitable network. Wi-Fi is the usual transport, although supported devices and carrier configurations may permit backup over cellular data. An iCloud backup is not a complete mirror of the device.

Apple distinguishes between iCloud backup and data that synchronizes independently through iCloud. iCloud backup principally protects information and settings that are not already synchronized to another iCloud service. For example, when Messages in iCloud, iCloud Photos, iCloud Drive, or Notes synchronization is enabled, that synchronized content is generally maintained in its respective service rather than duplicated in the daily device backup.

Forensic acquisition must therefore consider both the device backup and separately synchronized iCloud services. The contents available depend on the user's configuration, the application's backup settings, the state and date of the backup, the authentication method, and the account's encryption configuration. Under standard data protection, Apple retains access to keys for some categories. When ADP is enabled, iCloud backup and many additional categories are end-to-end encrypted, and Apple does not possess the keys required to decrypt them.

Types of data in iCloud backups

iCloud backups often include a broad selection of user-generated and system data. In many cases, these backups contain data unavailable through local logical acquisition.

Common iCloud Backup contents may include the following:

- Device settings, Home screen layout, and application organization
- Application data that is eligible for backup and is not stored separately in iCloud
- SMS, MMS, and iMessage content when Messages in iCloud are not enabled
- Photos and videos when iCloud Photos are not enabled
- Call history and supported visual voicemail information
- Apple Watch backup data associated with the iPhone
- Other on-device information required to restore the device

Data already synchronized through services such as iCloud Photos, Messages in iCloud, iCloud Drive, Contacts, Calendars, or Notes are generally not duplicated in the device backup. Its availability must be evaluated through the corresponding synchronized service.

Note

It is crucial to note that some categories, especially iMessages, health data, HomeKit data, and iCloud drive files, may no longer be included in the backup if they are synced separately via an end-to-end encrypted iCloud service.

Accessing iCloud backups

There are three primary methods for you to access iCloud backups:

- **Using Apple account credentials:** When credentials are lawfully available, a supported forensic tool may be able to authenticate to the account and acquire accessible cloud data. Authentication commonly requires a verification code from a trusted device or trusted telephone number. You must obtain and use that code within the applicable legal authority; "intercepting" a code should not be presented as a routine forensic step.
- **Using authentication material from a trusted system:** Some forensic tools may support account tokens or other authentication material recovered from a trusted computer or device. The availability and usefulness of this material are tool-, service-, account-, and version-dependent. Tokens may expire or be invalidated by password changes, account-security events, sign-outs, or changes to trusted-device status. Their recovery and use must be specifically authorized, documented, and validated.
- **Using legal process directed to Apple:** Subject to applicable law, jurisdiction, account identification, retention, and Apple's response policies, Apple may provide categories of account data that it retains and can decrypt. A request does not guarantee that an iCloud backup exists or that Apple can provide its protected contents. When ADP is enabled, Apple cannot decrypt end-to-end encrypted categories.

Note

Cloud authentication can be time sensitive. Authentication tokens may expire or become invalid after account changes, password resets, security events, or other changes to the trusted-device state. When lawfully available, document and preserve relevant authentication material promptly, while minimizing unnecessary interaction with the account.

Impact of ADP

Apple introduced **advanced data protection** (**ADP**) for iCloud in late 2022 as part of its ongoing effort to strengthen user privacy. Prior to ADP, most iCloud data was encrypted in transit and at rest on Apple's servers, but Apple still held the encryption keys for many categories. This meant that, with appropriate legal authority, Apple could decrypt and provide certain cloud-stored data to investigators. ADP fundamentally changes this model by extending end-to-end encryption to a much larger set of iCloud data types. Under ADP, only the user's trusted devices hold the decryption keys; Apple does not.

From a forensic standpoint, this shift is highly significant. When ADP is enabled, iCloud backup, Photos, Notes, iCloud Drive, Safari bookmarks, Voice Memos, and many additional categories become end-to-end encrypted. Apple cannot decrypt or provide protected content because the necessary keys remain under the control of the user's trusted devices. Some categories including Health data and passwords in iCloud Keychain are end-to-end encrypted even under standard data protection. Mail, Contacts, and Calendar remain under standard protection because they must interoperate with established internet protocols.

For investigators, ADP introduces new operational challenges. Possession of an Apple account password alone does not guarantee access to end-to-end encrypted cloud data. Access may additionally require two-factor authentication, an enrolled trusted device, the passcode or password of a trusted device, or an account-recovery mechanism. Two-factor authentication, recovery keys, trusted device enrollment, and the user's own device passcode all now play a more central role in determining whether encrypted data can be accessed. If ADP is active, forensic access typically requires a trusted device that is already enrolled in the user's iCloud account or possession of the necessary authentication secrets. Otherwise, access is effectively blocked by cryptography rather than policy.

This makes it essential for forensic examiners to determine early in the investigation whether ADP was enabled prior to or during device seizure. If ADP is active, acquisition strategies must shift accordingly by prioritizing on-device extraction, token recovery, or alternate data sources such as synchronized third-party cloud services, rather than relying on iCloud backups. ADP therefore represents one of the most impactful changes to iCloud forensic workflows since the introduction of default iOS encryption in 2014.

The following walkthrough demonstrates how you can use Magnet AXIOM to acquire available iCloud backup data for forensic examination.

Practical walkthrough: Acquiring an iCloud backup using Magnet AXIOM

The following walkthrough reflects the tested version of **Magnet AXIOM** shown in the figures. Cloud-acquisition interfaces, authentication requirements, service support, and available artifact categories change frequently. Before using this procedure, verify the vendor's current compatibility documentation and validate the workflow against a controlled account. Successful account authentication does not guarantee access to every iCloud service or to data protected by ADP.

Follow these steps to acquire iCloud backup using Magnet AXIOM:

1. Open **Magnet AXIOM** and go to **EVIDENCE SOURCES**:

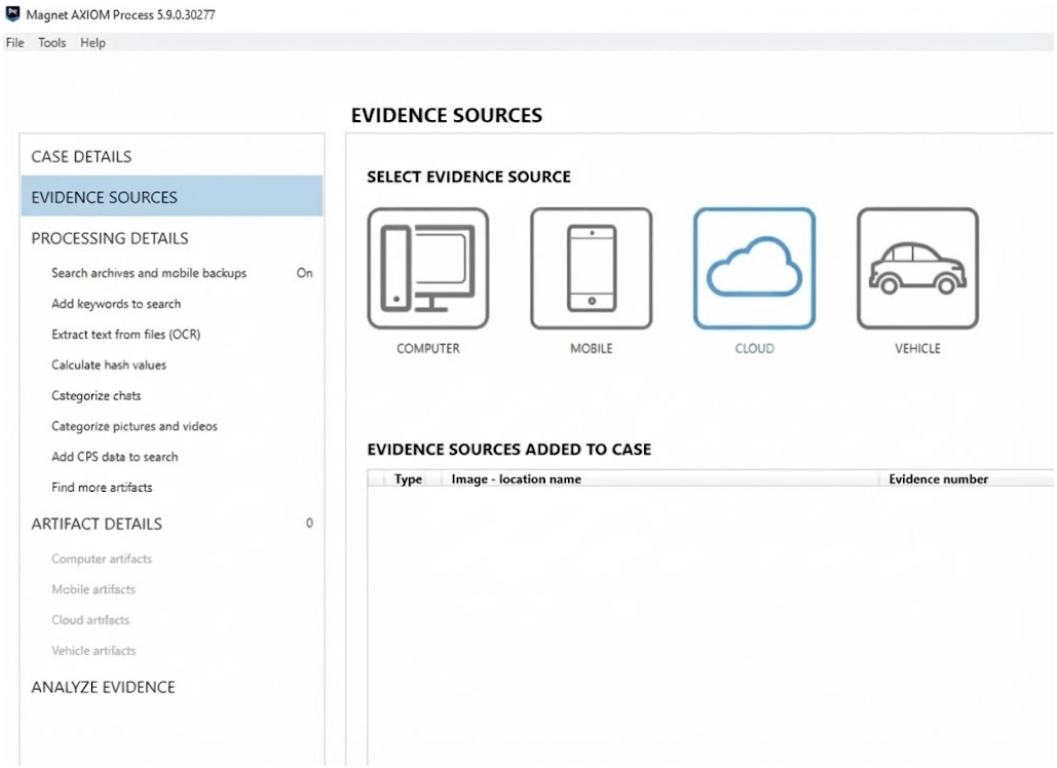


Figure 4.11: Magnet AXIOM Evidence Sources screen for adding a cloud-based evidence source

2. Select the **CLOUD** module to begin setting up a cloud-based acquisition:

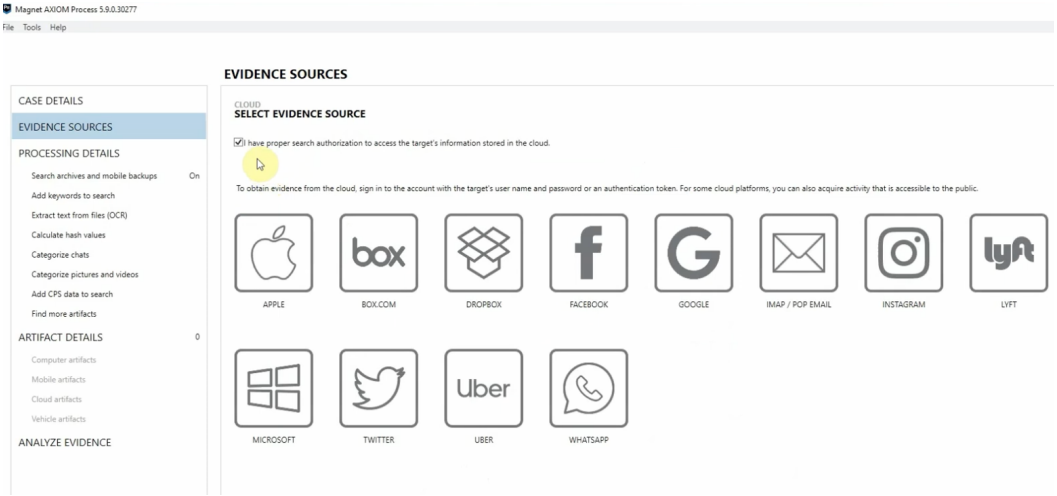


Figure 4.12: Magnet AXIOM cloud evidence-source selection screen showing supported services, including Apple

3. Click **Acquire Evidence** and choose **APPLE** as the cloud service
4. AXIOM presents two authentication options:
    - Sign in using Apple ID username/password
    - Use an authentication token extracted from a trusted device

    For this walkthrough, select the username/password option:

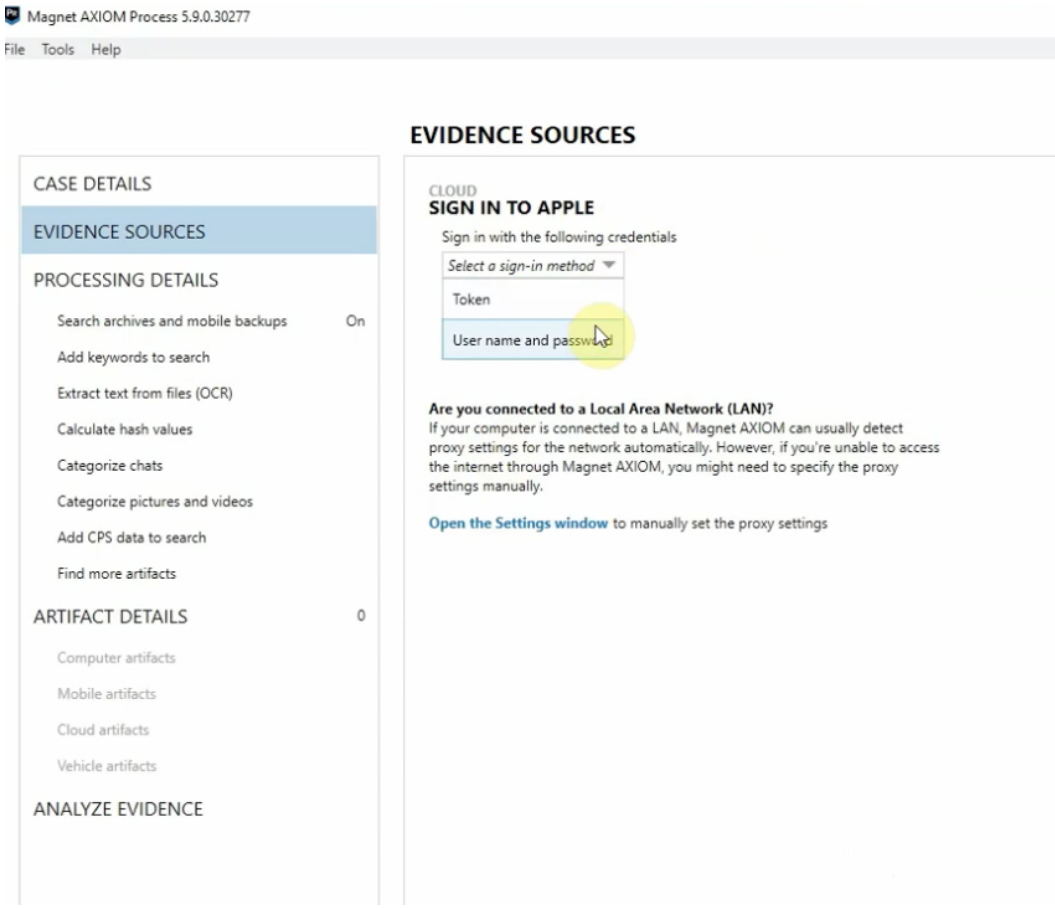


Figure 4.13: Magnet AXIOM Apple cloud acquisition sign-in screen with authentication method options

5. Enter the Apple ID credentials and click **SIGN IN**:

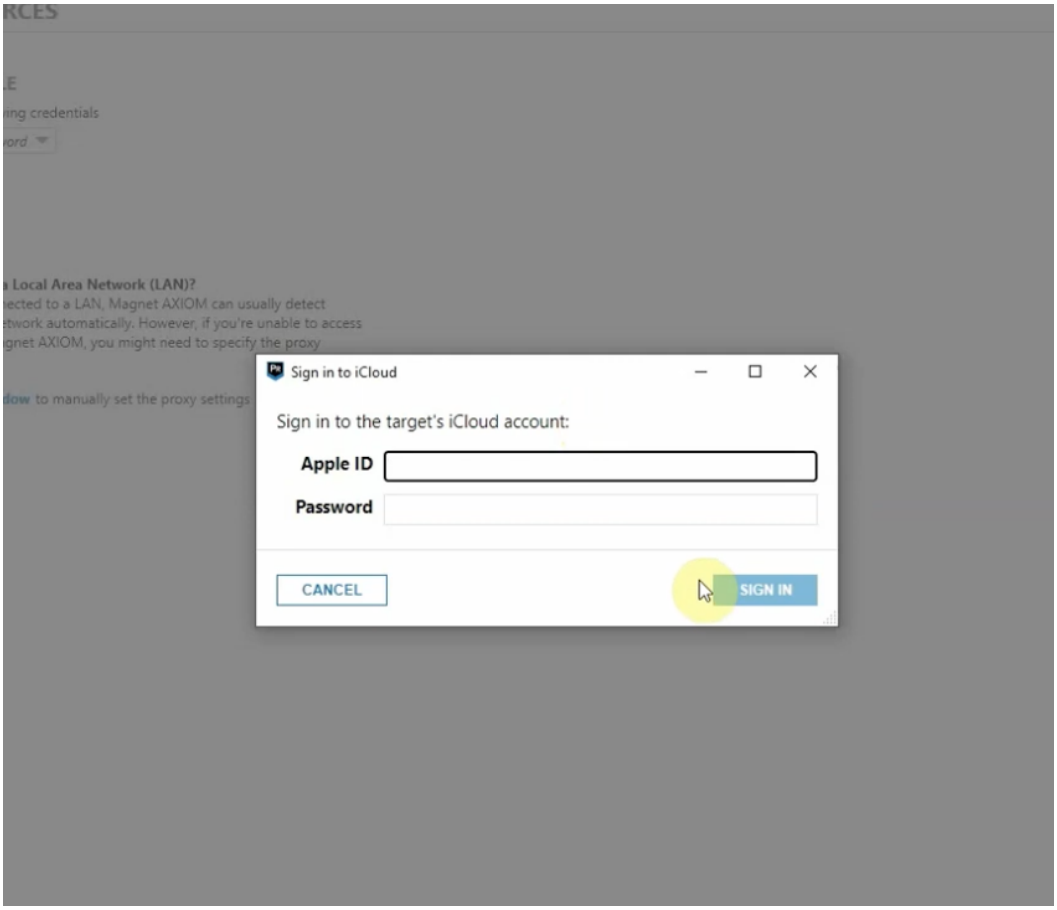


Figure 4.14: Magnet AXIOM prompt for entering the target iCloud account Apple ID and password

6. AXIOM will prompt for the multi-factor authentication (MFA) code associated with the Apple ID. Enter the verification code to continue:

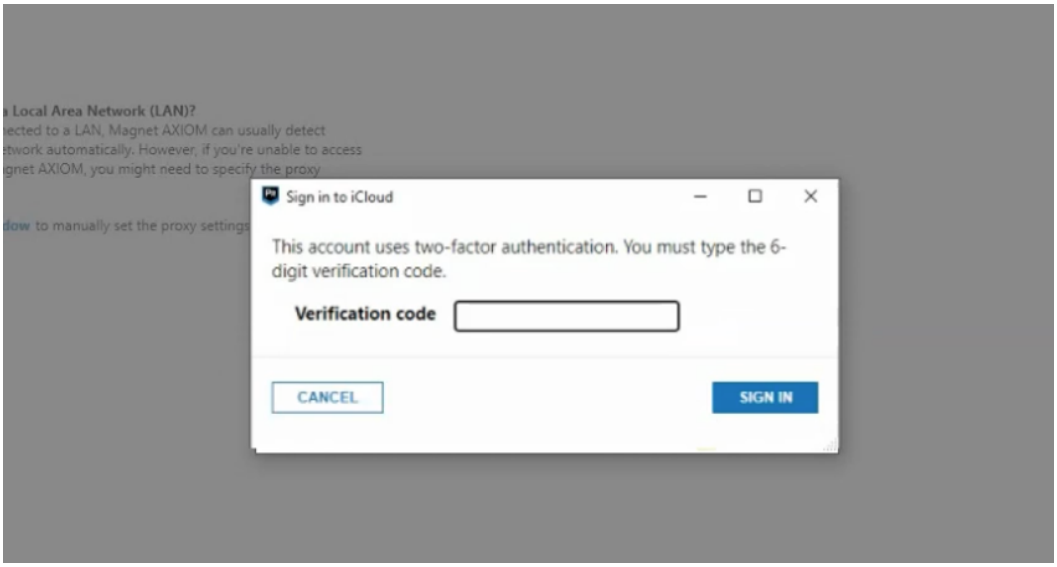


Figure 4.15: Magnet AXIOM prompt for the six-digit multi-factor authentication verification code

7. After successful authentication, AXIOM displays the available iCloud artifacts for acquisition. Review and select the items you want to collect.
8. Click **ANALYZE EVIDENCE** to begin downloading and processing the selected iCloud data

9. Once processing is complete, AXIOM presents the recovered artifacts such as browser history, call logs, messages, media files, and other synced data in the analysis interface:

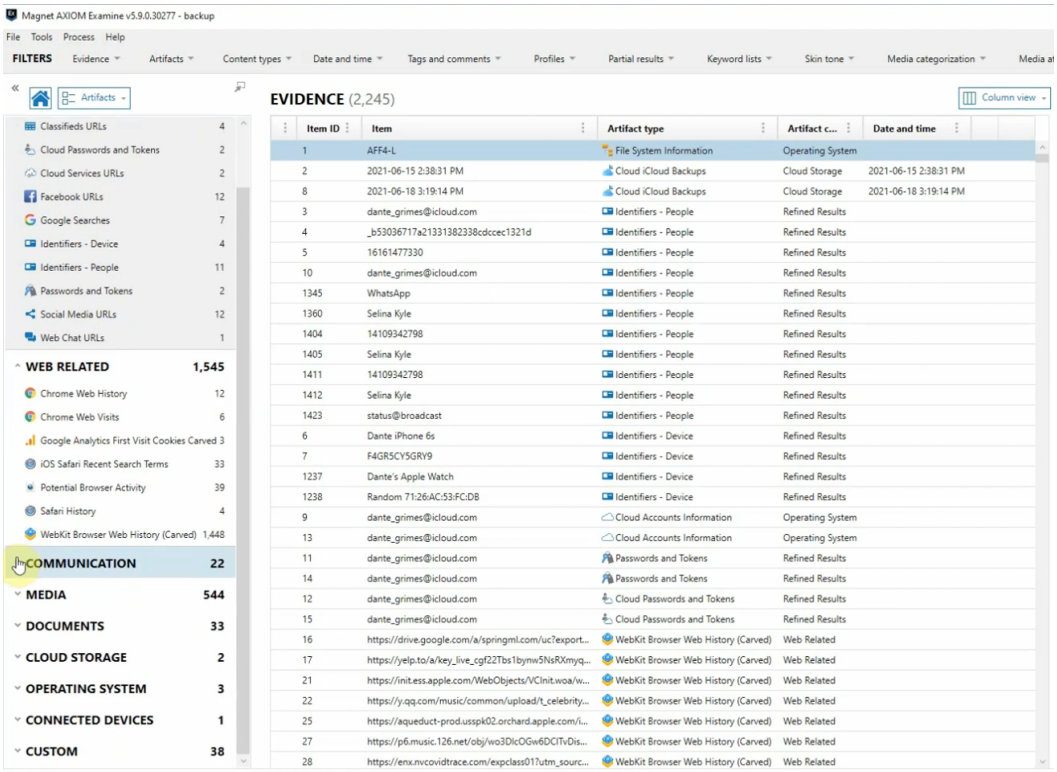


Figure 4.16: Magnet AXIOM Examine displaying artifacts recovered from the acquired iCloud backup

This walkthrough demonstrated how iCloud backup data can be acquired and examined using a forensic tool. The artifacts available will depend on the user's iCloud configuration, enabled services, authentication method, and whether ADP is enabled. You should therefore treat iCloud backups as one potential source of evidence and correlate recovered artifacts with local backups, device data, and other available sources.

Summary

This chapter examined how iOS backups (both local and cloud-based) serve as essential evidence sources, especially on modern devices where full file system acquisition is often restricted by Apple's security architecture. Backups function as structured snapshots of device data and frequently contain valuable artifacts such as messages, photos, app data, configuration files, and device identifiers. We reviewed how local backups created through iTunes/Finder are stored on macOS and Windows, and how key metadata files (`Info.plist`, `Manifest.plist`, `Status.plist`, and `manifest.db`) describe their contents. The chapter highlighted the significant difference between unencrypted and encrypted backups, with encrypted backups providing access to sensitive items such as Keychain data and health information. Practical examples showed how forensic tools generate and parse these backups.

We then explored iCloud backups, emphasizing that Apple distributes data across multiple iCloud services and increasingly uses end-to-end encryption. Features such as ADP further restrict cloud-based evidence unless proper credentials and authority are obtained. Walkthroughs demonstrated how commercial forensic tools authenticate and retrieve available iCloud artifacts. Finally, we covered common limitations such as selective backups, storage constraints, and Apple's evolving privacy design while reinforcing that backups remain one of the most reliable acquisition methods for newer iPhones.

The next chapter will shift from acquisition to analysis, focusing on how to interpret databases, timestamps, application data, and deleted artifacts to build meaningful forensic findings.

5

iOS Data Analysis and Recovery

In the previous chapters, we focused on the acquisition of data from iOS devices, including filesystem extractions, logical acquisitions, and backup-based collection from both local computers and iCloud. While acquisition is a critical step, it represents only the initial part of a forensic investigation. The true value of mobile forensics lies in the ability to interpret, validate, and contextualize the data that has been collected.

An iOS filesystem image or backup can contain thousands of files, databases, logs, and configuration records. Modern forensic tools parse much of this data automatically, presenting timelines, conversations, and media in a readable format. However, automated parsing alone is not sufficient in many investigations. Tools may miss artifacts, misinterpret timestamps, fail to decode proprietary structures, or overlook deleted and residual data. In some cases, you must manually examine raw files to validate findings, recover additional evidence, or explain how conclusions were reached.

This chapter focuses on the analysis phase of iOS forensics. It explains how data is structured and stored on iOS devices, how to interpret common timestamp formats, and how to work directly with SQLite databases and property list files that underpin most iOS artifacts. You will learn where key evidence is stored, how different applications record user activity, and how to recover deleted or partially overwritten data when possible.

The chapter also emphasizes an important investigative mindset: forensic tools are aids, not authorities. Understanding how artifacts are generated and stored allows you to recognize gaps, resolve inconsistencies, and defend your methodology in court. This is especially important as Apple continues to evolve iOS with stronger encryption, privacy protections, and system-level intelligence that can affect what data is available and how it should be interpreted.

In this chapter, we will cover the following topics:

- Foundations of iOS data interpretation
- Understanding iOS time and metadata
- SQLite databases in iOS: Structure and forensic relevance
- Locating SQLite databases in iOS evidence
- Core artifacts: Analyzing key iOS database files
- Deleted data recovery and limitations

By the end of this chapter, you should be comfortable moving beyond surface-level tool output and into deeper, defensible analysis of iOS evidence. The analytical principles covered here apply across filesystem extractions, local backups, and cloud-derived datasets, although the artifacts and level of access available will differ between acquisition sources.

Foundations of iOS data interpretation

Acquiring data from an iOS device is only the first step in a forensic investigation. The real value of evidence emerges during analysis, where raw data is interpreted, correlated, and validated. This phase is often underestimated, particularly by new practitioners who assume that clicking "parse" or "export" in a forensic tool is sufficient. In reality, **iOS data analysis** is rarely a push-button exercise, and blind reliance on tool output can lead to missed artifacts, incorrect conclusions, or evidence that fails under legal scrutiny.

Modern forensic tools are excellent at surfacing large volumes of data quickly. For example, they can automatically extract thousands of messages from `sms.db`, reconstruct Safari browsing history from its underlying databases, identify application artifacts across multiple directories, and correlate photographs with timestamps and location metadata. They decode databases, normalize timestamps, associate media with metadata, and present results in timelines and reports. However, tool output is an interpretation of evidence, not the evidence itself. Understanding this distinction is fundamental to sound forensic practice.

Parsed results vs. actual evidence

The **actual evidence** consists of the original files, databases, and records extracted from the device or backup. **Parsed results** are representations created by forensic software after applying assumptions, decoding logic, and parsing rules. These rules may change across tool versions, may not account for undocumented schema changes, or may fail silently when encountering corrupted or partially overwritten data.

For example, a forensic tool may display a list of SMS messages extracted from `sms.db`, but that view may exclude deleted records residing in freelists or write-ahead logs. Similarly, a tool may present a contact list without showing merged records, historical values, or artifacts created by

third-party apps that also write to contact databases. In such cases, relying only on parsed views risks overlooking critical information.

From a forensic standpoint, the database file itself is the evidence. The parsed table view is merely a convenience layer built on top of it.

Tool output and raw data validation

Validation is the process of confirming that what a tool reports accurately reflects what exists in the underlying data. Courts increasingly expect you to demonstrate not just what a tool found, but how and why it found it. This is especially important when findings are challenged or when the defense uses a different tool that produces conflicting results.

Raw data validation typically involves opening SQLite databases, property list files, or media files directly and confirming values manually. This may include verifying timestamps, confirming foreign key relationships between tables, checking deleted records in freelists, or examining metadata that a tool did not expose.

Manual validation is not an admission that forensic tools are unreliable. Rather, it demonstrates your competence and due diligence. An examiner who can explain how a timestamp was derived, where a record was stored, and why it is reliable carries significantly more credibility than one who can only reference a report generated by software.

Note

Why does manual analysis still matter in court?

In courtroom testimony, it's the examiners who need to testify, not the forensic tools. Judges and attorneys may ask how a message timestamp was calculated, why a record appears deleted but still recoverable, or whether a tool altered data during parsing. Without an understanding of how iOS stores and structures its data, these questions become difficult to answer convincingly.

Manual analysis also becomes essential when tools fail. Corrupt databases, partially overwritten files, unusual application behavior, or new iOS versions can all cause parsers to miss data. In such cases, the ability to fall back to direct database queries, hex inspection, or file-level analysis often makes the difference between a stalled investigation and a successful one.

Filesystem artifacts vs. backup-derived artifacts

Another critical foundation for analysis is understanding where the data came from. Artifacts extracted from a filesystem acquisition (**filesystem artifacts**) are not the same as artifacts

derived from a backup (**backup-derived artifacts**), even when they appear similar on the surface.

Filesystem acquisitions may contain live databases, caches, logs, temporary files, and application state that never make it into a backup. They may also include deleted or residual data within databases, caches, WAL files, and other application storage that may not be present in a backup. Backup-derived artifacts, on the other hand, reflect a curated snapshot created by Apple's backup process. Some data is excluded entirely, some are transformed, and some are re-encrypted using backup-specific key material.

You must always interpret findings considering their acquisition source. A missing artifact in a backup does not mean it never existed on the device. Conversely, a record present in a backup may reflect an earlier state that no longer exists on the device itself.

Effective iOS forensic analysis requires moving beyond tool-driven workflows and developing an evidence-driven mindset. Tools accelerate analysis, but understanding validates it. Throughout this chapter, we will examine how iOS stores data, where critical artifacts reside, and how to manually verify and recover information when automated parsing falls short. With this foundation in place, we can now move into the technical mechanics of iOS data interpretation, beginning with how time is represented across iOS artifacts.

Understanding iOS time and metadata

Time is the backbone of most mobile investigations. Whether you are reconstructing a conversation, proving a user's presence at a location, or establishing intent before an incident, nearly every conclusion depends on timestamps. iOS devices record time across a wide range of artifacts, including messages, calls, browser activity, application events, media metadata, and system logs. The challenge for examiners is that iOS does not rely on a single timestamp format. Multiple formats coexist across Apple applications and third-party apps, and the same extraction may contain several time representations side by side.

Most commercial forensic tools convert timestamps automatically and present them in readable form. However, automated conversion should be treated as a convenience, not a guarantee. Tools can misinterpret time values when the format is unusual, when the timestamp unit changes, or when artifacts are partially corrupted. For example, a forensic tool may display a message with a date in 1970 when the underlying timestamp is actually stored in milliseconds but interpreted as seconds. Without validating the raw value and its format, an examiner could place the artifact at the wrong point in the investigation timeline. For this reason, you must be able to recognize the most common timestamp formats used on iOS and validate conversions manually when needed.

The three timestamp formats most frequently encountered in iOS investigations are Unix time, Mac absolute time, and WebKit (or Chrome) time.

Unix timestamps

Unix time is one of the most widely used timestamp formats in computing. It represents the number of seconds that have elapsed since midnight on January 1, 1970 (UTC). Many iOS artifacts and third-party applications store timestamps in Unix format because they are simple, portable, and supported everywhere.

In practice, Unix timestamps appear in more than one unit. Some applications store Unix time in seconds, while others store it in milliseconds or nanoseconds. A timestamp stored in seconds may look like `1700000000`, while a timestamp stored in milliseconds will look like `1700000000000`. Without recognizing the unit, you may convert the value incorrectly and generate a date that is decades off.

Converting a Unix timestamp to a human-readable date on a Mac can be done directly in the Terminal using the `date` command as shown in *Figure 5.1*:

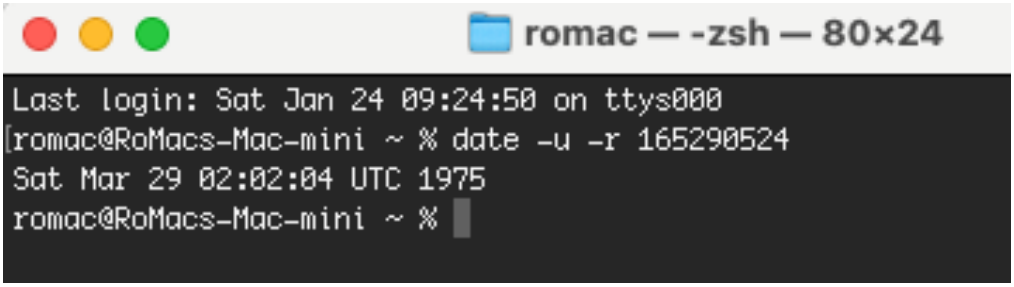


Figure 5.1: Converting Unix timestamp to human-readable date format on Mac

When validating Unix time, the simplest approach is to look at the magnitude of the number. Values with 10 digits typically represent seconds, while 13 digits often represent milliseconds. Larger values may represent nanoseconds. Once the unit is confirmed, conversion can be performed using local workstation utilities or known-good offline conversion methods. Correctly identifying and converting the timestamp format is particularly important when producing timelines that may be challenged, as you must be able to explain exactly how the time was derived.

Tip

When a timestamp is important to the investigation, validate the raw stored value before relying on the date displayed by a forensic tool. Confirm the timestamp format, unit, epoch, and time zone, particularly when correlating events across different applications.

Mac absolute time

Mac absolute time is heavily used by Apple across iOS system databases. Instead of measuring time from 1970, it uses a different epoch: midnight on January 1, 2001 (UTC). The value represents the number of seconds since that point in time.

This format appears frequently in artifacts generated by Apple applications such as call history, messages, Safari history, calendar, notes, and many system-level databases. Mac absolute time can be converted to Unix time by adding a fixed offset of 978,307,200 seconds. Once converted into Unix time, the value can then be interpreted in the same way as other Unix timestamps.

Converting a Mac absolute timestamp to a human-readable date on a Mac can be done directly in the Terminal. For example, in the following case, the Mac absolute timestamp of `759000000` is converted to human-readable format by initially adding `978307200` to it (which converts it to Unix time) and then using the same date command to convert it further to human-readable format:

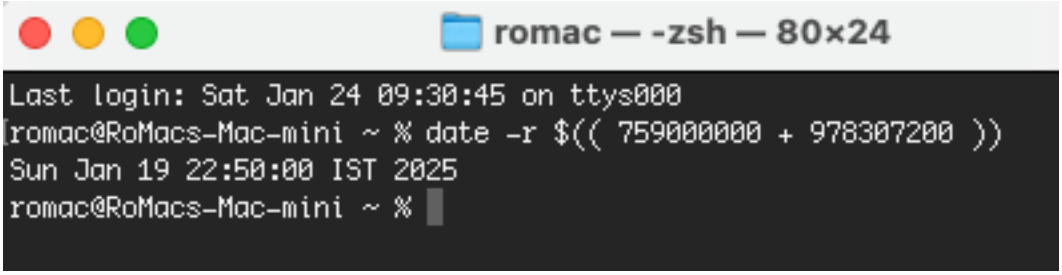


Figure 5.2: Converting Mac absolute timestamp to human-readable date format on Mac

Mac absolute time may also appear as a floating-point number rather than a whole integer. This often indicates sub-second precision and is common when timestamps are recorded as part of structured frameworks or system services. During validation, you should avoid rounding values prematurely, especially when attempting to correlate events that occur within seconds of each other.

WebKit and Chrome time

Browser artifacts often rely on a different timestamp model. WebKit and Chromium-based timestamp formats represent time as the number of microseconds since midnight on January 1, 1601 (UTC). This is sometimes referred to as **WebKit time** or **Chrome time** and is commonly encountered in Safari artifacts, Chrome browsing databases, cookies, and web-related caches.

On a Mac system, you can convert WebKit time to human readable format using the command shown in *Figure 5.3*:

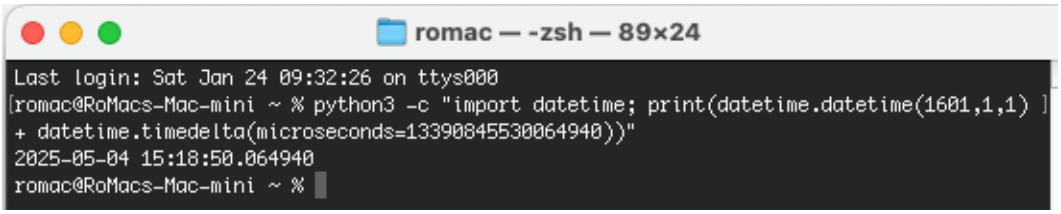


Figure 5.3: Converting WebKit timestamp to human-readable date format on Mac

Because the epoch is unfamiliar and the values are very large, WebKit timestamps are frequently misinterpreted. A common mistake is assuming the number is in Unix time, which results in impossible dates. When validating browser evidence, it is important to first confirm whether the artifact originates from a WebKit-based datastore and then apply the correct conversion logic.

WebKit timestamps are particularly important in investigations involving browsing behavior, link clicks, search activity, and web-based communications. Since browsers often serve as the bridge between applications, they can support or contradict other evidence found in messages or application databases.

Tip

Use a validated offline conversion method for evidentiary timestamps. Do not submit case values, identifiers, or other sensitive evidence to a public web service. Preserve the raw timestamp, conversion formula, tool or script version, time-zone assumption, and converted result so another examiner can reproduce the calculation.

Time zones, device settings, and interpretation errors

Correct timestamp conversion does not automatically guarantee correct interpretation. A timestamp may be accurate in raw form but still lead to incorrect conclusions if time zone context is not considered. Some iOS artifacts store timestamps in UTC regardless of the user's time zone, while others rely on device-local time at the moment of recording.

You should account for additional complications such as daylight-saving time shifts, manual user adjustments, travel-related time zone changes, or device settings that were altered during the relevant period. These factors can cause events to appear out of order when artifacts are merged into a single timeline.

When inconsistencies occur, the correct approach is not to force alignment but to document what the raw evidence shows and correlate additional artifacts. For example, the presence of multiple timestamps across messages, call history, and system usage data can help confirm whether an event truly occurred at the time shown or whether it reflects a configuration change.

Timestamp interpretation is one of the most common sources of forensic error because it often appears routine. However, when the outcome of an investigation depends on proving sequence and timing, timestamp mistakes become critical. Being able to confirm the raw timestamp format, apply correct conversions, and explain the process clearly is part of what separates a tool operator from a forensic examiner.

Once you understand how iOS timestamps work, the next step is to examine the storage systems where most of these values live. iOS relies heavily on SQLite databases, and forensic analysis often depends on reading them directly, validating tool output, and recovering residual records when possible.

SQLite databases in iOS: Structure and forensic relevance

SQLite databases are one of the most important building blocks in iOS forensics. While iOS devices store media files, caches, and application documents as regular files, a large portion of investigative evidence such as messages, call history, notes, browsing activity, and user interaction metadata is stored inside SQLite databases. These databases exist both in device filesystems and inside iOS backups, which makes them central to nearly every acquisition type discussed in earlier chapters.

As a forensic examiner working with iOS evidence, you must understand not only how to open and view SQLite databases, but also why SQLite behaves the way it does. Many of the most valuable findings in mobile investigations come from understanding database structure, recognizing when a tool has missed records, and recovering residual or deleted records that remain embedded in database internals.

Why iOS uses SQLite everywhere

SQLite is a lightweight, self-contained database engine designed to be embedded into applications. Unlike server-based database systems, SQLite does not require a separate process

to run, does not require network access, and stores everything in a single portable file. This design makes it ideal for mobile operating systems where power, storage, and performance constraints are strict. iOS uses SQLite extensively because it provides a reliable way to store structured data efficiently while allowing fast access by the operating system and applications.

Apple's own system apps rely heavily on SQLite. Messages, call history, calendar events, and Safari history all depend on database storage. Third-party applications follow the same pattern, using SQLite to store chat histories, cached content, authentication records, and user activity traces. As a result, SQLite analysis is not an optional skill, but one of the most practical areas where manual examination can add value beyond what automated tools provide.

Examining SQLite databases in practice

SQLite databases on iOS typically have common file extensions such as `.db`, `.sqlite`, or `.sqlitedb`, but they may also exist without any extension at all. The most common approach for manual analysis is to open the database using a reliable viewer such as DB Browser for SQLite. This provides a structured view of tables and allows you to run targeted queries:

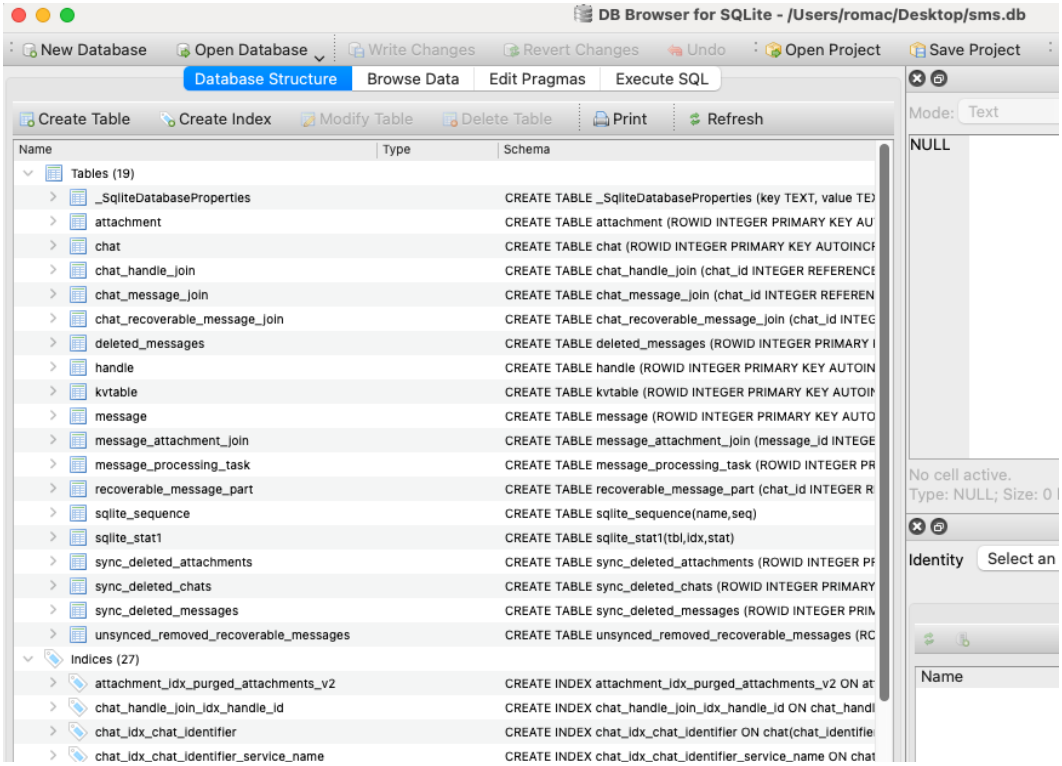


Figure 5.4: Viewing sms.db file using DB Browser for SQLite (DB4S)

If you prefer command-line workflows, the **sqlite3** tool allows direct querying of the database. This is especially useful when working in controlled forensic environments, scripting automated analysis, or validating results without relying on a GUI. Regardless of the method used, you should always work from a copy of the evidence. Original database files should remain untouched and preserved as part of the chain of custody.

Working with SQLite databases

At a simple level, an SQLite database can be thought of as a file containing multiple tables. Each table stores rows and columns like a spreadsheet, but with stronger structure. A database file can also contain indexes that improve performance, views that act like saved queries, and triggers that execute actions automatically when data changes. The most relevant elements are tables and the relationships between them, since tables hold the records that represent real-world activity such as calls, messages, visited URLs, or saved contacts.

In SQLite databases, the data is organized into tables that store structured records such as messages, call logs, contacts, and application activity. To view and interpret this data, you need a tool capable of opening SQLite database files. Most commercial mobile forensic suites, including Belkasoft Evidence Center, Magnet AXIOM, and Cellebrite UFED Physical Analyzer, support SQLite examination and often provide additional parsing and recovery features.

If you do not have access to commercial tools, there are several reliable free utilities available that can be used to open and analyze SQLite databases manually:

- **DB Browser for SQLite (DB4S)**: `http://sqlitebrowser.org/`.
- **fqlite**: A forensic SQLite recovery tool designed to recover deleted records and other residual data from SQLite databases, including content from freelists and related database structures. `https://www.staff.hs-mittweida.de/~pawlaszc/fqlite/`
- **SQLite command-line client**: `http://www.sqlite.org/`.
- **Letos**: A free, cross-platform SQLite database management tool that allows users to browse database structures, inspect records, and execute SQL queries. `https://letos.org/`
- **SQLiteSpy**: `https://www.yunqa.de/delphi/apps/sqlitespy/index`.

macOS includes the SQLite command-line utility, sqlite3, by default. This tool allows you to open SQLite database files directly and run queries against them from the Terminal. In the sections that follow, we will use both sqlite3 and graphical SQLite viewers to examine common iOS databases and extract relevant artifacts. Before we begin analyzing specific files, it is useful to understand a few basic commands that make navigating and querying SQLite databases easier.

Connecting to a database

Manual examination of iOS SQLite database files is possible with the use of free tools. The following is an example of how to examine a database using native Mac commands in the Terminal.

To connect to an SQLite database from the command line, run the sqlite3 command in the Terminal by entering your database file. This will give you an SQL prompt where you can issue SQL queries, as shown in the following code block:

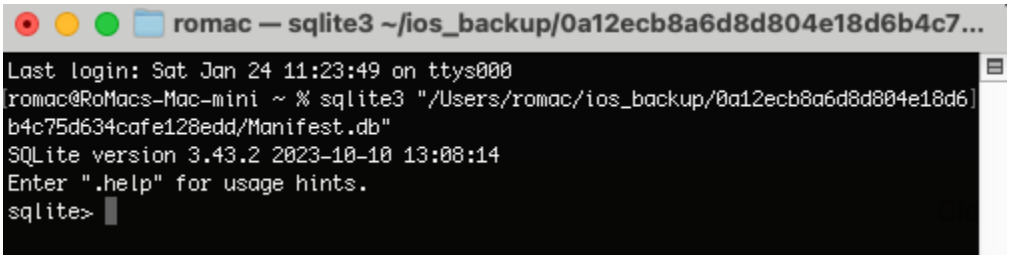


Figure 5.5: Connecting to a SQLite database

To disconnect, use the `.exit` command. This exits the SQLite client and returns to the Terminal. The next section will walk you through the analysis of databases, using sqlite3 built-in commands.

Exploring SQLite key commands

Once you connect to a database, you can use a number of built-in SQLite commands that are known as dot commands and that can be used to obtain information from the database files. You can obtain the list of special commands by issuing the `.help` command in the SQLite prompt. These are SQLite-specific commands, and they do not require a semicolon at the end. The most commonly used dot commands include the following:

- `.tables`: This lists all the tables within a database. The following screenshot displays the list of tables found inside the `sms.db` database:



Figure 5.6: SQLite command to see list of tables

- `.schematable-name`: This displays the SQL CREATE statement that was used to construct the table. *Figure 5.7* displays the schema for the handle table from the `sms.db` database:

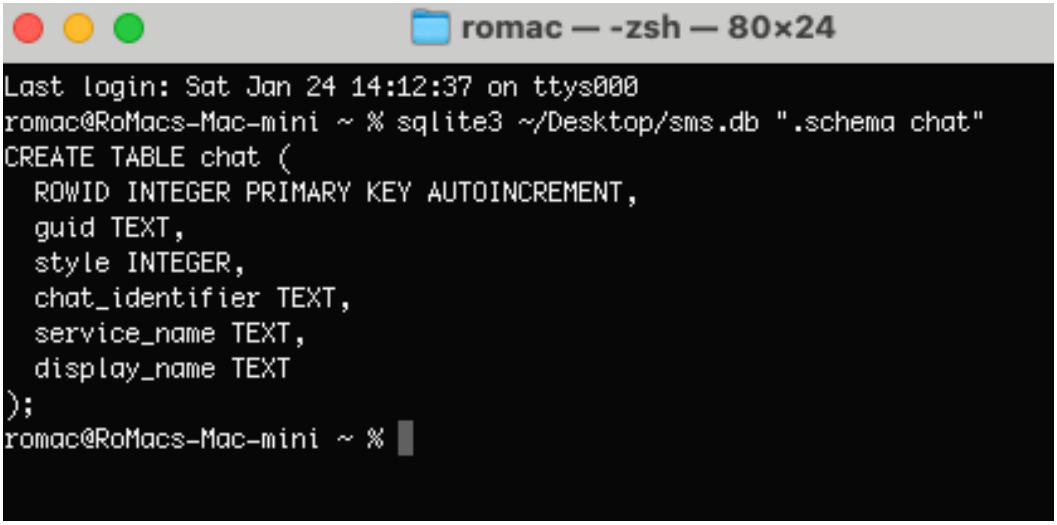


Figure 5.7: SQLite command and SQL statement used for the table

Tip

When examining an unfamiliar SQLite database, inspect the schema before querying its records. Table names, columns, relationships, and indexes can reveal how the application organizes its data and help identify relevant artifacts that automated parsers may not expose.

- `.dump table-name`: This dumps the entire content of a table into SQL statements. The example in the following screenshot displays the dump of the handle table, which is found inside the `sms.db` database:

```
);
romac@RoMacs-Mac-mini ~ % sqlite3 ~/Desktop/sms.db ".dump chat"
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE chat (
  ROWID INTEGER PRIMARY KEY AUTOINCREMENT,
  guid TEXT,
  style INTEGER,
  chat_identifier TEXT,
  service_name TEXT,
  display_name TEXT
);
INSERT INTO chat VALUES(1,'iMessage;+;chat1',45,'+919999000111','iMessage','Frie
nd A');
INSERT INTO chat VALUES(2,'SMS;+;chat2',43,'+919888000222','SMS','Friend B');
COMMIT;
romac@RoMacs-Mac-mini ~ %
```

Figure 5.8: SQLite command that dumps the entire content of a table into SQL statements

- `.output file-name`: This redirects the output to a file on the disk instead of showing it on the screen
- `.headers on`: This displays the column title whenever you issue a `SELECT` statement
- `.help`: This displays the list of available SQLite dot commands
- `.exit`: This disconnects from the database and exits the SQLite command shell
- `.mode`: This sets the output mode; it could be `.csv`, `HTML`, tabs, and so on

Along with SQLite's built-in dot commands, you can also run standard SQL statements directly against a database from the command line. For forensic examination, `SELECT` statements are generally the most relevant because they allow you to inspect records without intentionally modifying the database. Unlike dot commands, SQL statements must end with a semicolon. Most iOS databases you examine will contain a manageable number of records, so a simple `SELECT *` query is often enough to display all rows in a table for review. We will use these queries throughout this chapter as we analyze common iOS artifacts.

How SQLite evidence is parsed and why tools can differ

Most forensic tools present SQLite artifacts in easy-to-read reports. This makes analysis fast, but it can also mask complexity. Tools make decisions about what to parse, which tables matter, how to interpret timestamps, and whether to include deleted records or not. Two different tools analyzing the same database may produce different results because they implement different parsing logic or prioritize different tables.

This is why you must treat tool output as one interpretation of the database, not the definitive truth. When results matter, you should validate the database manually. This may involve opening the database in an independent viewer, confirming table schemas, verifying time conversion logic, and checking whether relevant records exist outside what the tool reported.

Manual validation also becomes crucial when tools fail silently. A tool may skip a database because the schema is newer than expected, because the file is partially corrupted, or because a field is stored differently. In such cases, manually opening the database may reveal that evidence still exists but was not interpreted correctly.

Filesystem extraction vs. backup-based SQLite evidence

The same SQLite database may appear different depending on how it was acquired. Databases extracted from a filesystem acquisition are often the "live" copies used by the device at that moment. These may include WAL files, temporary files, caches, and the most recent records. Databases extracted from backups may be older snapshots and may not include companion files in the same way. In some cases, the backup process normalizes, re-packages, or excludes certain artifacts, which can affect how much residual evidence is available for recovery.

You must always interpret SQLite findings in context. A database record found in a backup may represent activity from days or weeks earlier. A record missing from a backup does not prove the activity never happened, only that it may not have been included in that backup snapshot. When multiple sources exist, comparing databases across sources can reveal changes over time, including deletions, edits, and shifts in user behavior. With these principles in mind, the next step is to understand where SQLite databases are typically located within iOS evidence and how their location differs between filesystem extractions and backups.

Locating SQLite databases in iOS evidence

A significant portion of iOS forensic analysis revolves around examining SQLite databases. Messages, call history, browser activity, application state, and many system artifacts are stored in SQLite format. However, the process of locating these databases differs depending on whether you are working with a filesystem extraction or an iOS backup.

In filesystem acquisitions, filenames, and directory structures are preserved but data is distributed across sandbox containers. In contrast, iOS backups rename files using a hash-based storage model that requires metadata reconstruction before meaningful analysis can begin. Understanding both approaches is essential to ensure that important databases are not overlooked.

The following sections describe how SQLite databases can be located in each acquisition scenario.

Locating SQLite databases in filesystem extractions

In filesystem acquisitions, original filenames and directory structures remain intact, which makes identifying SQLite databases more intuitive. The primary challenge instead comes from iOS sandboxing, where artifacts are distributed across application containers, shared service directories, and system domains.

Each installed application operates within its own sandbox container located under `/private/var/mobile/Containers/Data/Application/`. Inside this directory, applications are identified using randomly generated UUID folder names rather than bundle names. As a result, you must first determine which container belongs to a specific application before locating its SQLite databases.

In practice, this mapping can be achieved by examining container metadata files, application bundle identifiers, or by using forensic tools that automatically resolve UUID-to-application relationships.

Once the correct container is identified, SQLite databases are typically found within common subdirectories such as the following:

- `Documents/` for persistent user data
- `Library/` for application databases, preferences, and caches
- `tmp/` for temporary or transient storage

System and built-in application databases are stored outside app containers, commonly within these paths:

- `/private/var/mobile/Library/`
- `/private/var/mobile/Library/SMS/`
- `/private/var/mobile/Library/CallHistoryDB/`
- `/private/var/mobile/Library/Safari/`

Because filenames are preserved, you can locate SQLite databases using keyword searches for extensions such as `.db`, `.sqlite`, or `.sqlitedb`. However, some databases may lack extensions, making content-based identification necessary.

It is also important to remember that filesystem acquisitions often include associated WAL and SHM files. These companion files may contain recent or deleted records that are not present in the main database and should always be examined alongside the primary SQLite file.

Filesystem analysis relies more heavily on understanding iOS containerization and application storage patterns. Familiarity with common artifact paths significantly reduces analysis time and helps prevent important databases from being overlooked.

Locating SQLite files from iOS backups

The process of locating SQLite databases in iOS backups differs substantially from filesystem analysis. When a backup is created using Finder or iTunes, files are not stored using their original filenames. Instead, Apple uses a hash-based storage model in which each file is renamed using a SHA1-derived identifier. This design improves efficiency and integrity but requires investigators to reconstruct metadata before analysis.

Backups are typically stored on macOS at `~/Library/Application Support/MobileSync/Backup/`. Within this directory, each device backup appears as a folder named after the device identifier. Inside the backup folder, files are stored using long hexadecimal filenames, making direct identification of SQLite databases impossible without referencing metadata.

This mapping is performed using `Manifest.db`, which is itself a SQLite database. Opening `Manifest.db` in a tool such as DB Browser for SQLite allows you to view the Files table, which maps hashed filenames to their original iOS paths. Key fields include the file identifier, domain, and relative path, which together reveal the original location of the artifact on the device.

Once the relevant entry is identified, you can retrieve the actual database from the backup directory. Files are organized into subfolders based on the first two characters of the file identifier, with the complete hash serving as the filename. After copying the file to a working directory, it can be renamed appropriately and opened for examination.

This process must be repeated for each artifact of interest when working with backup evidence. Although the mapping step introduces additional complexity, backups often contain extensive historical data that may no longer be present on the live device.

The following steps outline how to identify and extract the relevant SQLite databases for analysis:

1. The first step is to locate the backup folder on macOS. This is easy as the backups are commonly stored at: `~/Library/Application Support/MobileSync/Backup/`

2. Inside this backup folder, files are renamed as 40-character SHA1 hashes and are not directly readable. In other words, you cannot identify SQLite files just by filename. You must first map them using the `Manifest.db` file.
3. Fortunately, `Manifest.db` itself is an SQLite database and can be opened using DB Browser for SQLite (DB4S) or any SQLite-compatible tool
4. After opening `Manifest.db`, navigate to the **Browse Data** tab in DB4S and select the **Files** table. This table contains metadata that maps the hashed filenames to their original iOS locations:

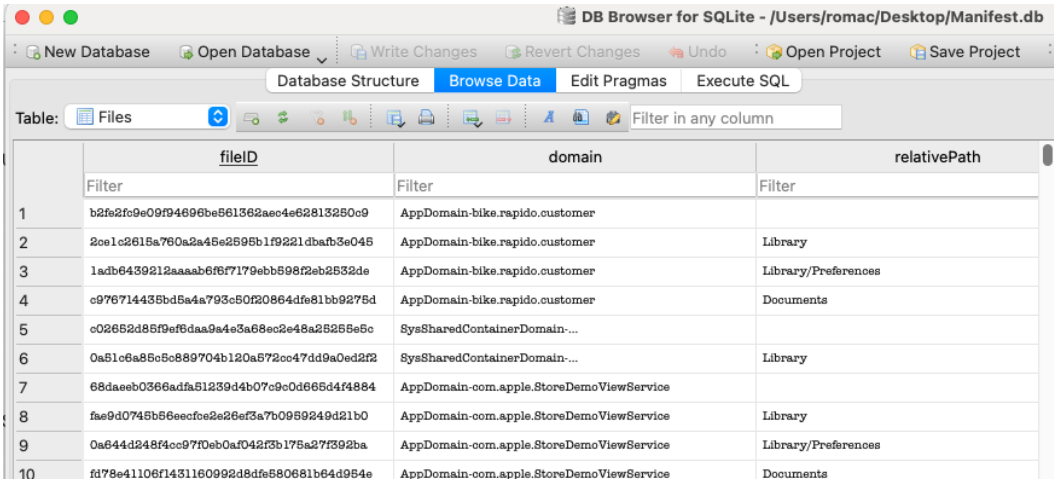


Figure 5.9: Examining the Files table in Manifest.db using DB Browser for SQLite

The table has the following columns:

- **fileID**: this is the actual filename
- **domain**: this is the app/system container
- **relativePath**: this is the original iOS path

For example, consider the following figure:

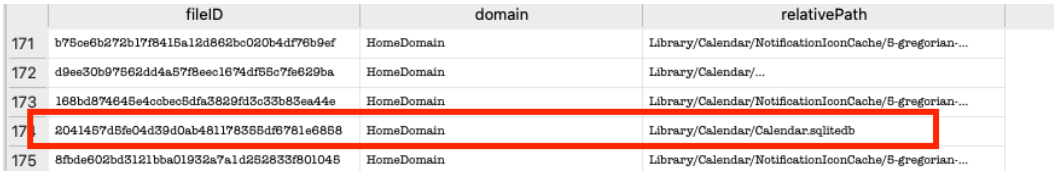


Figure 5.10: Locating SQLite databases in an iOS filesystem extraction

The table shows that the original file path is `Library/Calendar/Calendar.sqlitedb` and the corresponding fileID is `20414...`, this indicates that the Calendar database has been renamed to that SHA1 value 20414... within the backup.

5. To retrieve the actual SQLite file, return to the backup directory. Files are stored in subfolders named after the first two characters of the fileID. Therefore, if the fileID begins with 20, the file will be located inside the folder named 20, with the full 40-character SHA1 value as its filename:

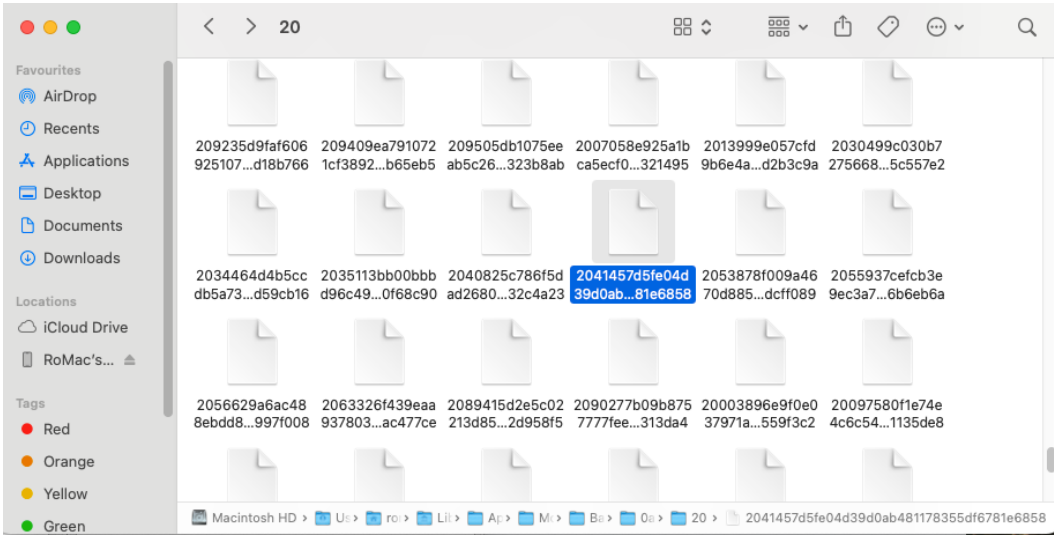


Figure 5.11: Locating a backup file using its SHA-1 fileID

Tip

When locating an artifact through `Manifest.db`, also check for related files from the same original directory. Examining only the primary database may overlook companion databases, configuration files, attachments, or other artifacts needed to interpret the evidence completely.

6. Copy the file to a separate working directory, rename it appropriately (for example, `calendar.db`), and open it using DB Browser for SQLite. You will then be able to view the database structure and browse its contents:

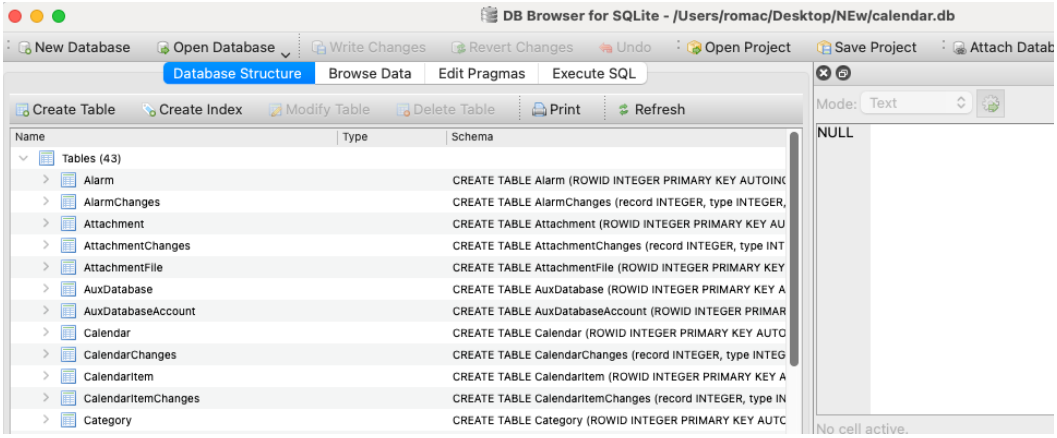


Figure 5.12: Examining the structure of calendar.db using DB Browser for SQLite

This same methodology can be applied to locate other SQLite artifacts within the backup. The following sections will examine several high-value databases commonly encountered during iOS forensic analysis.

Core artifacts: Analyzing key iOS database files

Once you understand how iOS timestamps work and how SQLite databases store structured records, the next step is identifying which artifacts matter most during an investigation. iOS devices generate an enormous volume of data, but not all of it is equally useful. A strong investigator learns to prioritize artifacts that answer investigative questions directly, such as who the user communicated with, what actions they performed, where they were, and how their activity changed over time.

Core iOS artifacts generally fall into two broad categories. The first category includes **communication and content artifacts** such as messages, calls, contacts, notes, and browser history. These sources often provide direct evidence of conversations, intent, planning, and relationships. The second category includes **behavioral and system artifacts** such as application usage patterns, background activity, and system intelligence records, which help confirm user presence, sequence of events, and credibility of timelines.

While many forensic tools automatically parse these artifacts into readable reports, you should understand where the original data resides and what it represents. This becomes especially important when a tool produces incomplete results, when key artifacts must be validated manually, or when multiple sources must be correlated.

The filesystem and backup data acquired in earlier chapters should contain several key SQLite databases that form the foundation of most iOS investigations. In the following sections, we will examine these databases, explain their forensic relevance, and demonstrate how they can be located and interpreted across both filesystem and backup acquisitions.

> **Note**
>
> The examples shown in the following sections were obtained from a device running iOS 16.7.12 using different acquisition methods. Apple frequently modifies database structures and storage locations across releases, so examiners should expect minor variations depending on the device model, iOS version, and acquisition method used. These sections will use examples obtained from both backup and filesystem acquisitions for illustration.

Contacts artifact: AddressBook.sqlitedb

The `Contacts` database is one of the most informative artifacts present on an iOS device. It contains detailed information about the user's personal and organizational relationships, including names, phone numbers, email addresses, notes, account associations, and metadata reflecting when contacts were created or modified. In many investigations, this artifact helps establish communication networks, identify associates, and correlate interactions across messaging platforms.

On modern iOS devices, contact information is primarily stored in the following database: `HomeDomain/Library/AddressBook/AddressBook.sqlitedb`. During filesystem analysis, this database can be accessed directly at its original path. When working with backups, however, the file must be located through `Manifest.db` (as per process described earlier), since backup storage replaces original filenames with SHA1 identifiers. After identifying the correct fileID, the corresponding database can be copied, renamed, and examined using a SQLite viewer.

The easiest way to identify the `Contacts` database within a backup is by opening `Manifest.db` and running the following query:

```
SELECT fileID, domain, relativePath FROM Files WHERE relativePath LIKE
'%AddressBook.sqlitedb%';
```

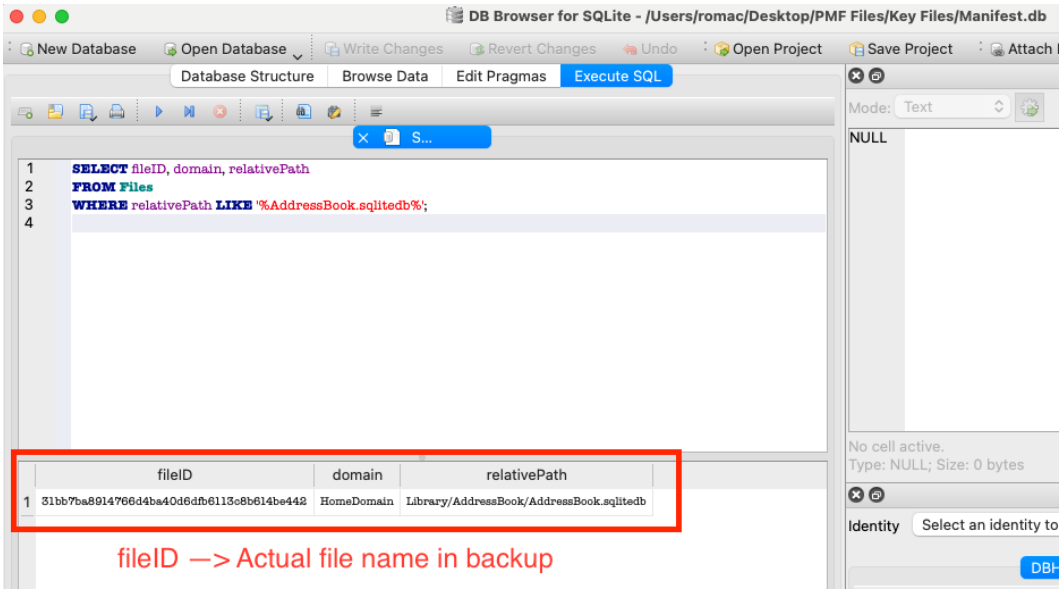


Figure 5.13: Locating AddressBook.sqlitedb in an iOS backup using Manifest.db

The query in *Figure 5.13* will return the corresponding fileID (the SHA1 filename used in the backup), along with the original iOS path. The same can be followed while identifying other sqlite files such as `sms.db`, `history.db`, etc.

The `AddressBook.sqlitedb` database contains several tables, of which the following are particularly important:

- **ABPerson**: This table contains core contact information, including first name, last name, organization, notes, and various metadata fields. Each contact record is uniquely identified by its ROWID.
- **ABMultiValue**: This table stores multi-value attributes associated with contacts, such as phone numbers, email addresses, URLs, and other communication details. The `record_id` field links each entry to the corresponding contact in the `ABPerson` table via the contact's ROWID.
- **ABMultiValueLabel**: This table maps label identifiers to human-readable values (for example, "Mobile," "Home," or "Work") that describe the type of data stored in the `ABMultiValue` table.

It is important to recognize that not all contact information resides exclusively within this database. Messaging and social networking applications frequently maintain their own contact caches inside application containers. As a result, you should review relevant third-party app directories to ensure that contact relationships derived from messaging platforms are not overlooked.

Recent iOS versions have also expanded synchronization and account-level metadata within the `Contacts` database. Additional tables may reflect iCloud synchronization state, linked accounts, or contact unification behavior.

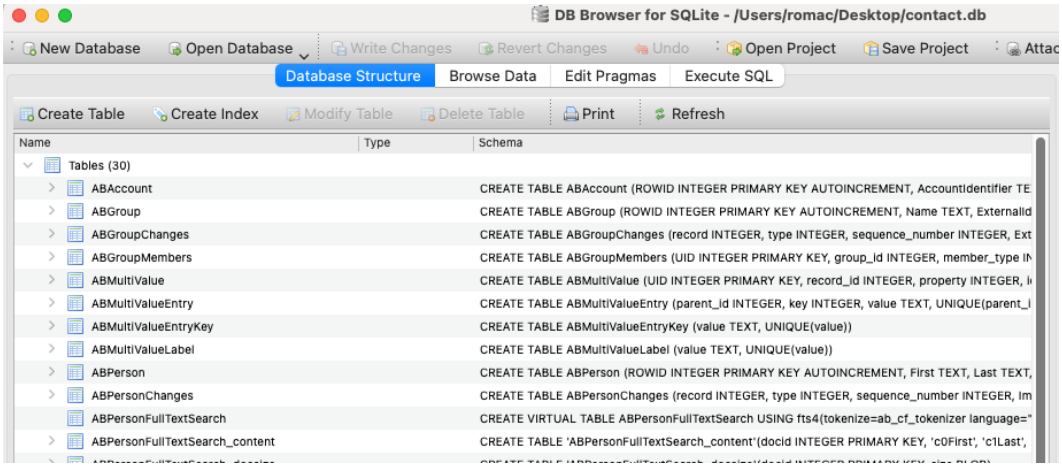


Figure 5.14: Examining the structure of contact.db using DB Browser for SQLite

Furthermore, encrypted backups must be decrypted before this database can be accessed, as certain contact metadata is protected by data protection classes.

While command-line SQLite analysis remains possible, tools such as DB Browser for SQLite provide a clearer interface for browsing tables, executing queries, and exporting results. These tools are particularly useful when combining multiple tables to reconstruct meaningful contact views.

To generate a consolidated list of contact names alongside their associated communication values, investigators can join the `ABPerson` and `ABMultiValue` tables using the following query:

```
SELECT
 ABPerson.First,
 ABPerson.Last,
 ABMultiValue.value
FROM ABPerson
LEFT JOIN ABMultiValue
ON ABPerson.ROWID = ABMultiValue.record_id;
```

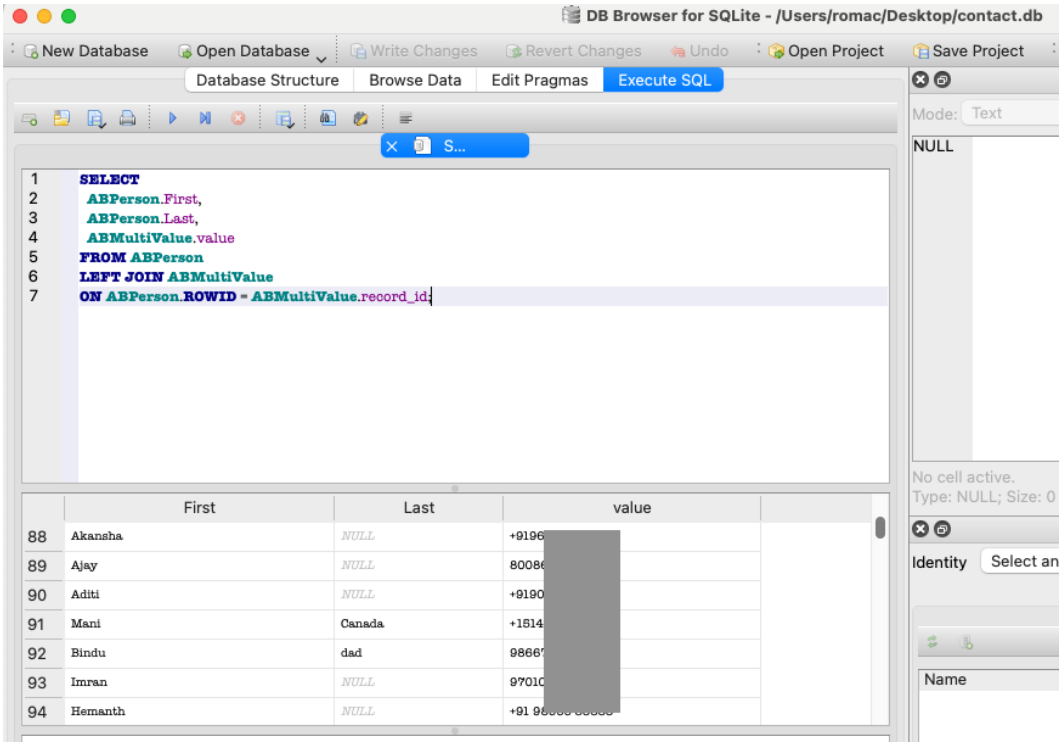


Figure 5.15: Querying contact names and phone numbers

This query produces a unified view that links each contact with its associated phone numbers or email addresses, allowing you to quickly identify key relationships and communication endpoints.

Call history artifact: CallHistory.storedata

The **call history artifact** captures cellular calls, FaceTime sessions, and, in many cases, supported third-party VoIP communications made or received by the user. This database is highly valuable during investigations because it helps establish communication patterns, identify contacts of interest, reconstruct timelines, and corroborate user activity across multiple evidence sources.

On modern iOS devices, call metadata is stored primarily in `CallHistory.storedata`. In a filesystem extraction, the database can typically be found at `/private/var/mobile/Library/CallHistoryDB/CallHistory.storedata`. In an iOS backup, the same artifact is represented under the `HomeDomain` domain as `Library/CallHistoryDB/CallHistory.storedata` and can be located through `Manifest.db`, which maps the backup file identifier to its original domain and relative path.

The `CallHistory.storedata` database uses Apple's core data schema. The `ZCALLRECORD` table contains the majority of communication artifacts. Although Apple occasionally modifies supporting fields across iOS releases, the primary structure remains stable.

Important columns commonly examined include the following:

- **ZDATE**: Stores the call timestamp in Mac Absolute Time and must be converted to a human-readable format
- **ZDURATION**: Represents the call duration in seconds
- **ZADDRESS**: Contains the phone number, Apple ID identifier, or VoIP handle associated with the call
- **ZSERVICE_PROVIDER**: Indicates the service used (cellular, FaceTime, or certain VoIP providers), helping differentiate call types
- **ZLOCATION**: May include carrier or regional metadata depending on network context and device configuration
- **ZORIGINATED / ZANSWERED**: Provide additional context regarding call direction and whether the call was answered or missed

Retention within the active database is limited, and older records may be removed as new entries are created. This does not necessarily indicate immediate deletion; remnants may persist within WAL files or residual database pages depending on device usage and acquisition method.

Investigators should also be aware that unencrypted backups may contain incomplete telephony metadata, whereas encrypted backups and filesystem extractions typically provide more comprehensive call records.

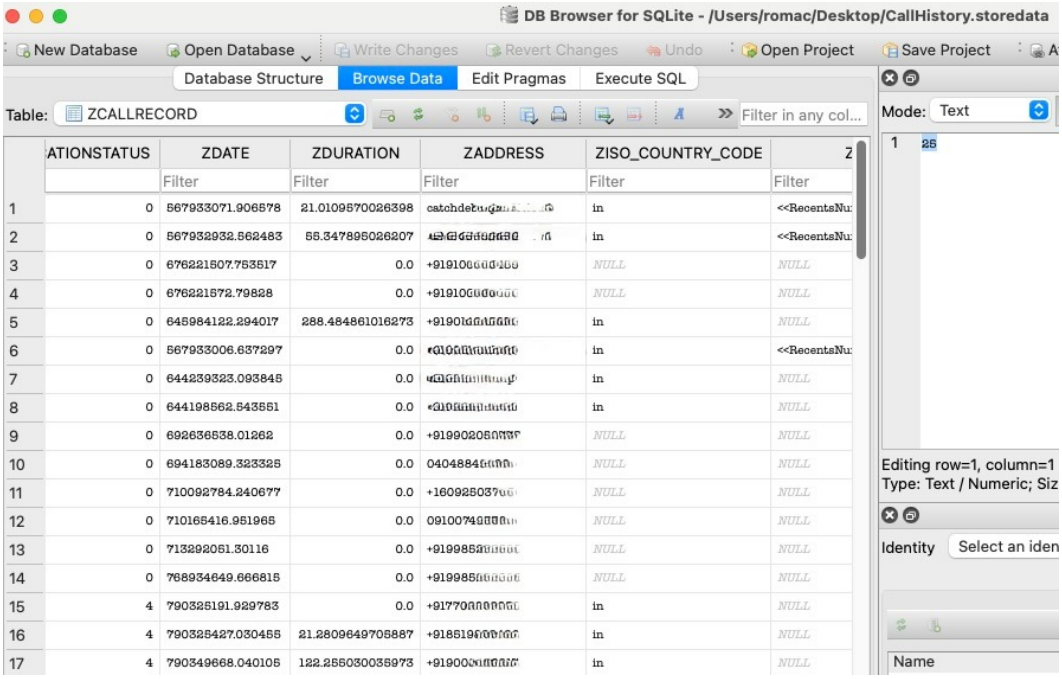


Figure 5.16: Examining call history records in the ZCALLRECORD table of CallHistory.storedata

You can run the following query in DB Browser for SQLite (DB4S) to display call history in a human-readable format. The query converts the Mac Absolute Time timestamp into local time and extracts key metadata such as duration, call direction, and answering status:

```
SELECT
datetime(ZDATE + 978307200, 'unixepoch', 'localtime') AS call_time,
ZDURATION AS duration_seconds,
CASE ZORIGINATED
WHEN 1 THEN 'Outgoing'
ELSE 'Incoming'
END AS direction,
```

```
CASE ZANSWERED
WHEN 1 THEN 'Answered'
ELSE 'Missed or Declined'
END AS answered,
ZADDRESS AS contact_identifier
FROM ZCALLRECORD
ORDER BY ZDATE DESC;
```

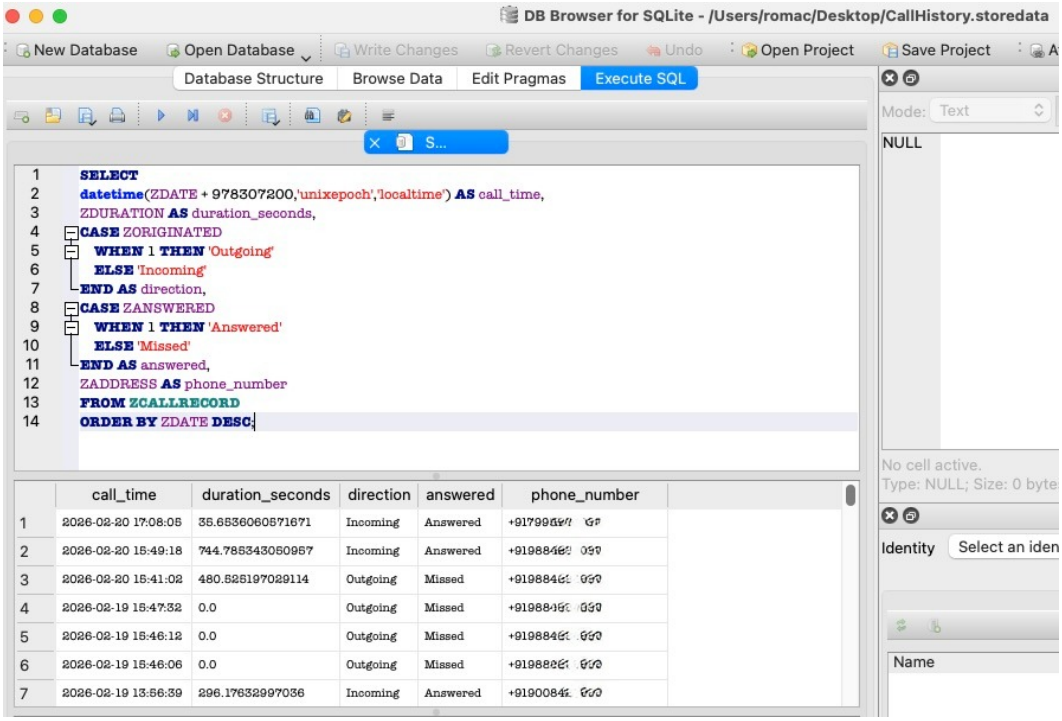


Figure 5.17: Examining CallHistory.storedata in DB4S

Because iCloud synchronization can replicate call history across Apple devices signed into the same Apple ID, investigators should also consider the possibility that call artifacts may appear on multiple devices or within backups even if the original device is unavailable.

Messages artifact: Sms.db

The `sms.db` database stores SMS, MMS, and iMessage communications exchanged by the device. This artifact is often one of the most important sources of evidence, as it contains message content, participant identifiers, timestamps, attachment references, delivery status, and conversation metadata. While SMS and MMS messages are transmitted through the cellular network, iMessage allows users to exchange messages over Wi-Fi or cellular data with other Apple users. From a forensic perspective, both communication types are stored within the same database, enabling you to reconstruct conversations across messaging channels.

On modern iOS devices, message data is primarily stored in sms.db. In a filesystem extraction, the database can typically be found at `/private/var/mobile/Library/SMS/sms.db`. In an iOS backup, the same artifact is represented under the `HomeDomain` domain as `Library/SMS/sms.db` and can be located through `Manifest.db`, which maps the backup file identifier to its original domain and relative path.

The `sms.db` database contains several tables, with message, handle, chat, and attachment being the most relevant. The message table stores the content and timestamps, while the handle table identifies communication participants.

A commonly used query to extract recent messages in readable form is shown here:

```
SELECT
    datetime(message.date/1000000000 + 978307200, 'unixepoch') AS readable_date,
    handle.id AS contact,
    message.text
FROM message
LEFT JOIN handle ON message.handle_id = handle.ROWID
ORDER BY message.date DESC
LIMIT 50;
```

This query converts Apple's timestamp format into human-readable time and associates each message with the corresponding participant:

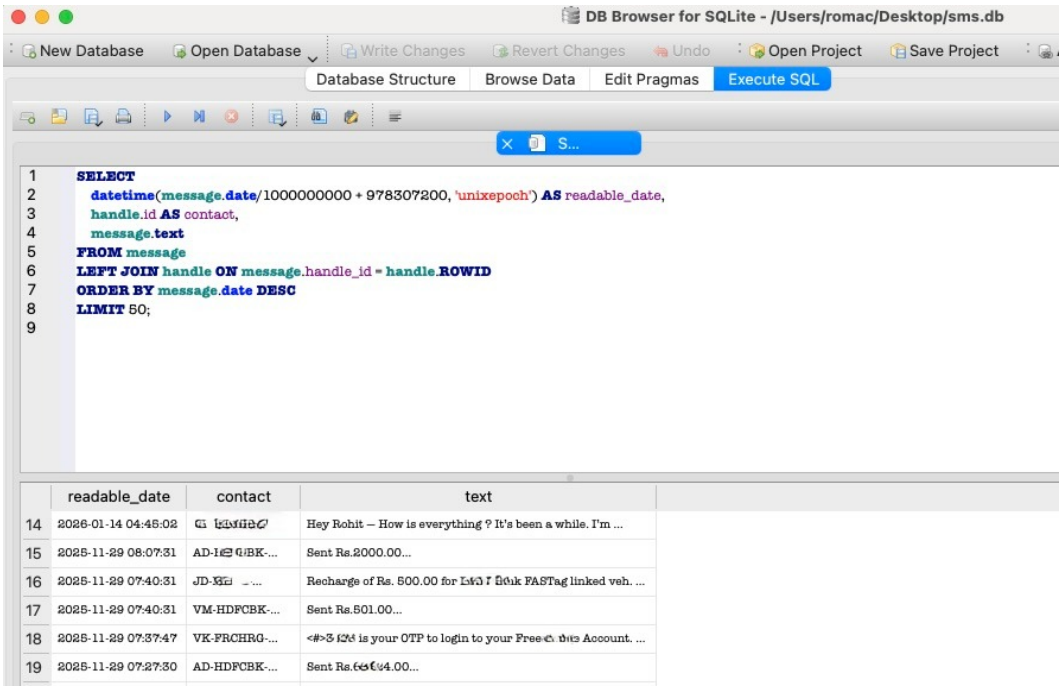


Figure 5.18: Examining sms.db in DB4S

Investigators should note that message timestamps may appear in different units (seconds, milliseconds, or nanoseconds) depending on iOS version and table usage. Careful validation of timestamp conversion is therefore essential.

An additional artifact of interest exists within the following directory: `HomeDomain/Library/SMS/Drafts/`. This location may contain `composition.plist` files representing unsent or draft messages created by the user. These artifacts can provide valuable context regarding user intent, partially composed communications, or messages deliberately abandoned before sending. Because message attachments are stored separately and referenced through the database, you should also correlate `sms.db` records with the attachments directory to obtain a complete view of conversations.

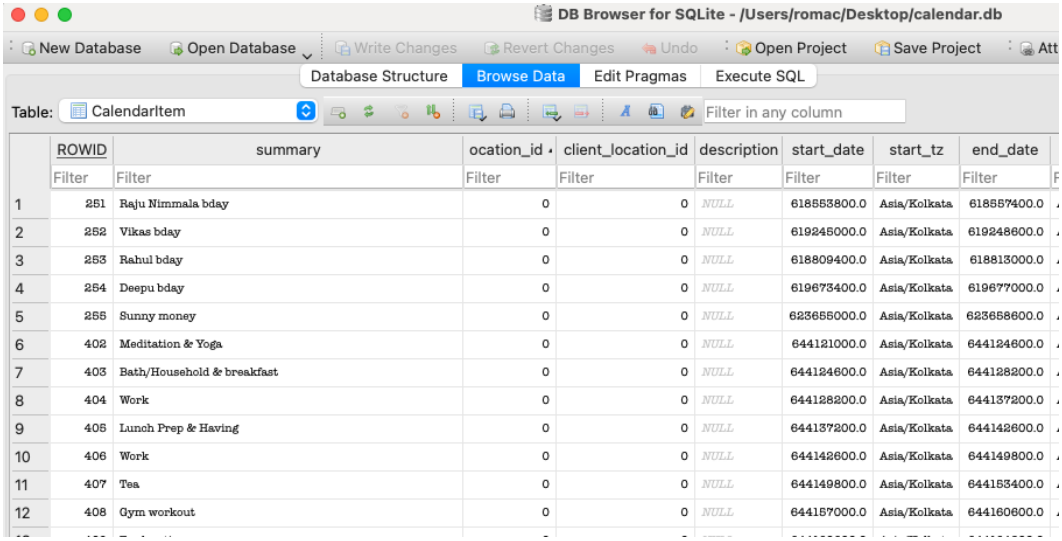
Calendar events artifact: Calendar.sqlitedb

Calendar data provides valuable insight into a user's planned activities, appointments, reminders, and meeting coordination. Events may be created manually on the device or synchronized from external sources such as iCloud, Exchange, Google, or other third-party

calendar services. From an investigative perspective, **calendar artifacts** can help reconstruct timelines, corroborate presence at meetings, and reveal intent or coordination with other individuals.

On modern iOS devices, calendar data is primarily stored in the following database: `/HomeDomain/Library/Calendar/Calendar.sqlitedb`. This database is typically available in both filesystem acquisitions and backups. When working with backups, the database must be located using `Manifest.db`, since backup files are renamed using SHA1 identifiers. Mapping the original path via the `Files` table allows the examiner to retrieve and analyze the correct database.

The `CalendarItem` table contains the core event information, including event summary, notes, start and end timestamps, recurrence data, and associated metadata such as location and organizer details. Depending on the synchronization source, additional tables may store attendee information, alarms, and calendar account relationships:



	ROWID	summary	ocation_id	client_location_id	description	start_date	start_tz	end_date
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	251	Raju Nimmala bday	0	0	NULL	618553800.0	Asia/Kolkata	618557400.0
2	252	Vikas bday	0	0	NULL	619245000.0	Asia/Kolkata	619248600.0
3	253	Rahul bday	0	0	NULL	618809400.0	Asia/Kolkata	618813000.0
4	254	Deepu bday	0	0	NULL	619673400.0	Asia/Kolkata	619677000.0
5	255	Sunny money	0	0	NULL	623655000.0	Asia/Kolkata	623658600.0
6	402	Meditation & Yoga	0	0	NULL	644121000.0	Asia/Kolkata	644124600.0
7	403	Bath/Household & breakfast	0	0	NULL	644124600.0	Asia/Kolkata	644128200.0
8	404	Work	0	0	NULL	644128200.0	Asia/Kolkata	644137200.0
9	405	Lunch Prep & Having	0	0	NULL	644137200.0	Asia/Kolkata	644142600.0
10	406	Work	0	0	NULL	644142600.0	Asia/Kolkata	644149800.0
11	407	Tea	0	0	NULL	644149800.0	Asia/Kolkata	644153400.0
12	408	Gym workout	0	0	NULL	644157000.0	Asia/Kolkata	644160600.0

Figure 5.19: Examining calendar event records in the CalendarItem table of Calendar.sqlitedb

Calendar timestamps are commonly stored in Mac Absolute Time format and must be converted to Unix time for human-readable interpretation. The following query can be executed in DB Browser for SQLite or a similar tool:

```
SELECT
    summary,
    datetime(start_date + 978307200, 'unixepoch') AS start_time,
    datetime(end_date + 978307200, 'unixepoch') AS end_time,
```

```
    description
FROM CalendarItem
ORDER BY start_date DESC
LIMIT 50;
```

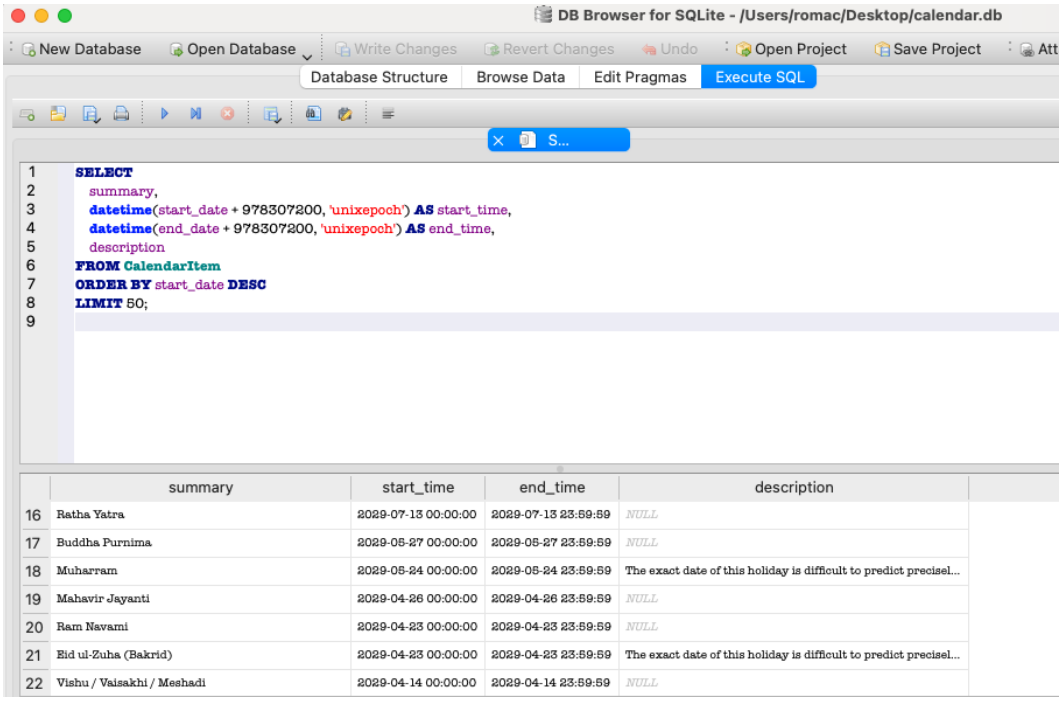


Figure 5.20: Examining calendar.sqlitedb in DB4S

Calendar artifacts are generally present in both encrypted and unencrypted backups. However, the context surrounding events may differ depending on the acquisition method. Encrypted backups and filesystem extractions often preserve richer synchronization metadata, attendee relationships, alarm triggers, and auxiliary tables that assist in reconstructing event provenance and cross-device activity. Filesystem acquisitions may also include associated WAL files and caches that reveal recently modified or deleted events not visible in the primary database.

You should also be aware that calendar events frequently originate from cloud synchronization. As a result, deletion or modification on one device may propagate across other devices and backups, making timeline comparison between multiple backups particularly valuable.

Notes artifact: NoteStore.sqlite

The **notes artifact** contains user-created content from Apple's built-in Notes application. While the interface appears simple, Notes frequently stores highly sensitive information such as credentials, financial details, personal journaling, investigation-relevant planning notes, scanned documents, and embedded media. Because users often treat Notes as a trusted personal repository, this artifact can provide unusually direct insight into intent, memory, and contextual behavior.

Earlier iOS versions stored Notes data in `/HomeDomain/Library/Notes/notes.sqlite`, where tables such as `ZNOTE` and `ZNOTEBODY` contained relatively accessible note titles, content, and timestamps. Modern iOS versions, however, introduced a significantly redesigned storage model based on core data and cloud synchronization.

On current iOS releases, the primary Notes database resides inside a shared app group container: `AppDomainGroup-group.com.apple.notes / NoteStore.sqlite`. This database is typically available in both filesystem extractions and backups. When working with backups, the database must be located using `Manifest.db`, since filenames are replaced with SHA1 identifiers.

Modern Notes app uses a core data schema in which note content is distributed across multiple interrelated tables. Metadata such as titles, creation dates, modification timestamps, and folder relationships are typically stored in `ZICCLOUDSYNCINGOBJECT`, while the actual note body resides within `ZICNOTEDATA.ZDATA` as a binary BLOB as shown here:

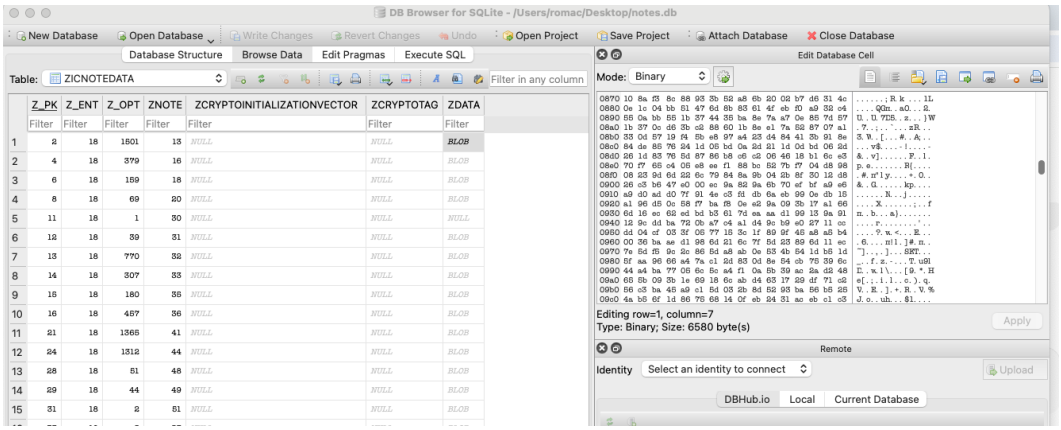


Figure 5.21: Examining notes in DB4S

Unlike legacy implementations, note content is no longer stored as plaintext. Instead, the **ZDATA** field contains compressed and serialized data structures that often use protobuf encoding combined with Apple attributed string formatting. As a result, note content cannot be directly interpreted through standard SQLite viewers. Meaningful reconstruction requires extracting the BLOB data, decompressing it when necessary, and decoding the structured representation before converting it into readable text.

To confirm the presence of notes within the database, you may execute:

- `SELECT COUNT(*) FROM ZICNOTEDATA`
- Metadata associated with notes can be enumerated using:

```
SELECT Z_PK, ZTITLE, ZCREATIONDATE, ZMODIFICATIONDATE
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZTITLE IS NOT NULL;
```

The exact metadata columns vary across iOS versions. Recent versions may contain fields such as ZTITLE1 and ZTITLE2, along with multiple creation and modification date fields. You should first inspect the schema of **ZICCLOUDSYNCINGOBJECT** to identify the fields present in the acquired database before querying note metadata. Full content recovery requires additional decoding of the `ZDATA` field, often supported by commercial forensic tools or specialized parsing scripts.

You should also consider that Notes frequently synchronize through iCloud. Consequently, note creation, modification, or deletion may propagate across multiple devices and backups. Comparing multiple snapshots can therefore reveal historical content, earlier revisions, or notes that were subsequently deleted from the device.

Safari bookmarks, history, and tab artifacts

The Safari browser on iOS stores a variety of artifacts that can help reconstruct browsing behavior, user intent, and interaction patterns. Even when users attempt to clear browsing activity, residual data may remain across multiple Safari databases and caches, making it important to examine these sources collectively rather than in isolation.

Bookmark data is stored in `/HomeDomain/Library/Safari/Bookmarks.db`. This database preserves bookmark titles, URLs, folder hierarchy, and ordering information. From a forensic perspective, bookmarks can reveal long-term interests, frequently accessed services, and potential evidence of planning or repeated activity, even when browsing history has been cleared.

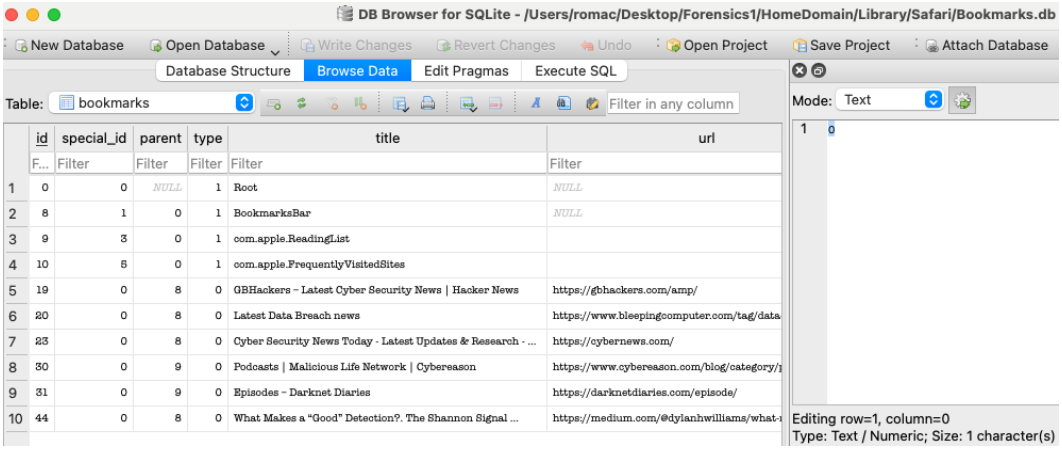


Figure 5.22: Examining bookmarks in DB4S

Browsing history is primarily maintained in `/HomeDomain/Library/Safari/History.db`. The `history_items` and `history_visits` tables typically contain visited URLs, visit counts, redirect chains, and timestamps. However, investigators may encounter situations where these tables appear partially populated or empty. This commonly occurs when a user manually clears history, uses private browsing mode, or when iCloud synchronization propagates deletion across devices. In such scenarios, additional Safari artifacts should be reviewed.

One particularly useful source is `/HomeDomain/Library/Safari/SafariTabs.db`. This database records open tabs, recently closed tabs, and session metadata. It may retain URLs, page titles, and tab state information even when corresponding entries are absent from `History.db`. Because Safari tab sessions can synchronize through iCloud, this artifact may also contain browsing references originating from other devices associated with the same Apple ID.

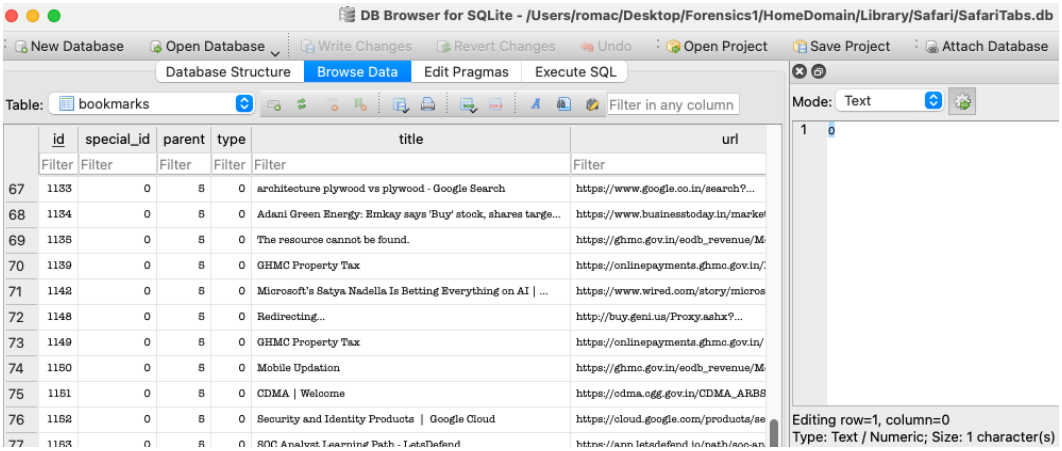


Figure 5.23: Examining Safari tab and browsing artifacts in SafariTabs.db using DB Browser for SQLite

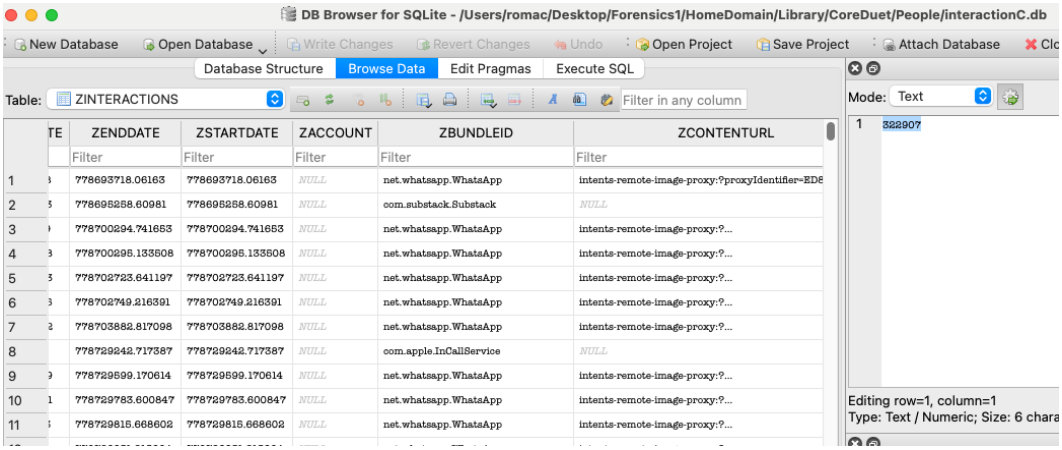
For a comprehensive reconstruction of Safari activity, you should correlate data from `Bookmarks.db`, `History.db`, `SafariTabs.db`, and relevant WebKit cache or session artifacts. Relying on a single database may lead to incomplete conclusions, particularly in cases involving history clearing, cross-device synchronization, or privacy-focused browsing behavior.

Device interaction and behavioral intelligence artifacts

iOS maintains behavioral intelligence databases that track how users interact with applications, contacts, and system features over time. One of the most valuable of these artifacts is `interactionC.db`, which forms part of Apple's CoreDuet framework. This framework supports Siri suggestions, proactive recommendations, and cross-application awareness, but from a forensic perspective it provides insight into user behavior that may not be visible in primary communication artifacts.

The database is typically located at `/HomeDomain/Library/CoreDuet/People/interactionC.db`. Within filesystem acquisitions, this path is preserved directly. In backup-based acquisitions, the file must be located through `Manifest.db` mapping due to Apple's hash-based storage structure.

The `ZINTERACTIONS` table records metadata describing user actions such as reading or sending messages, initiating calls, sharing content, launching applications, and interacting with contacts across different services. Rather than storing full message or call content, the database focuses on behavioral indicators including timestamps, bundle identifiers, associated contacts, interaction direction, and communication modality.



	TE	ZENDDATE	ZSTARTDATE	ZACCOUNT	ZBUNDLEID	ZCONTENTURL
		Filter	Filter	Filter	Filter	Filter
1	3	778693718.06163	778693718.06163	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?proxyIdentifier=ED8
2	3	778695258.60981	778695258.60981	NULL	com.substack.Substack	NULL
3	0	778700294.741653	778700294.741653	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
4	3	778700295.133508	778700295.133508	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
5	3	778702723.641197	778702723.641197	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
6	3	778702749.216391	778702749.216391	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
7	2	778703882.817098	778703882.817098	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
8		778729242.717387	778729242.717387	NULL	com.apple.InCallService	NULL
9	0	778729599.170614	778729599.170614	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
10	1	778729783.600847	778729783.600847	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...
11	3	778729815.668602	778729815.668602	NULL	net.whatsapp.WhatsApp	intents-remote-image-proxy:?...

Figure 5.24: Examining interactions in DB4S

Because this information reflects activity patterns rather than content, interaction artifacts may persist even when primary sources such as messages, call history, or application databases have been deleted or are partially unavailable. This makes the database particularly useful for identifying communication frequency, validating user presence, linking individuals to application usage, and strengthening timeline reconstruction.

In addition to the interaction records, examiners should review the `ZCONTACTS` table, which contains metadata describing contacts involved in recorded interactions. Correlating this table with other contact and communication artifacts can help confirm identities, reconstruct relationships, and support attribution during analysis.

User's phone numbers

Investigators can often identify phone numbers historically associated with a device by examining cellular usage databases that retain carrier and subscriber information across restores, SIM changes, and device migrations. Even when a user replaces a device and restores from backup, certain cellular identifiers may persist, providing insight into previously used numbers and network associations.

On modern iOS versions, this information is commonly stored in `WirelessDomain/Library/Databases/CellularUsage.db`. In filesystem extractions, the database can be accessed directly at this location. When working with iTunes or Finder backups, the file must be located through `Manifest.db`, since backup artifacts are stored using SHA-1–based filenames.

After extracting and opening the database in a SQLite viewer such as DB Browser for SQLite (DB4S), , investigators can examine subscriber-related tables (for example, `subscriber_info` or similarly named structures depending on the iOS version). These tables may contain phone

numbers, SIM identifiers (ICCID), carrier details, and other cellular metadata useful for attribution and device-to-subscriber correlation.

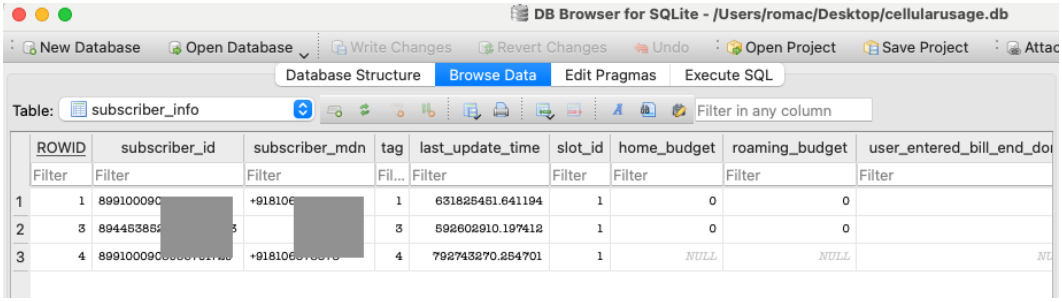


Figure 5.25: Examining phone details in DB4S

Because schema details may vary across iOS releases and carrier implementations, you should expect minor structural differences and review related telephony preference files and carrier configuration artifacts to corroborate findings.

Property lists (plist) and their forensic significance

Property lists, commonly known as **plist files**, are among the most widely used data storage formats across both iOS and macOS. They serve as structured containers for configuration settings, application preferences, device state information, and various forms of system metadata. While plist files typically do not contain large volumes of user-generated content, they often reveal critical contextual details about device configuration, user behavior, application usage, and system events. For this reason, they should not be overlooked during forensic analysis.

Plist files may exist in either XML (human-readable) or binary format. Modern iOS versions primarily rely on the binary format for efficiency and storage optimization. Although binary plists are not immediately readable, they can be safely converted into a human-readable representation without modifying the original evidence, allowing you to inspect their contents while preserving forensic integrity.

It is also important to recognize that plist artifacts do not always carry a `.plist` extension. Some files may appear without extensions or may be embedded within application containers, caches, or databases. Consequently, investigators must rely on directory context, internal structure, and tool-assisted identification rather than filename alone when locating relevant plist artifacts.

On macOS, the built-in plutil utility provides a simple way to inspect and convert plist files without installing additional software. In addition, most commercial forensic platforms automatically parse plist structures and present them in an interpreted format while retaining

access to the original binary representation for validation. When manual inspection is required, tools such as Xcode or dedicated plist viewers can also be used.

A commonly encountered example during backup and filesystem analysis is `com.apple.mobile.ldbackup.plist`:

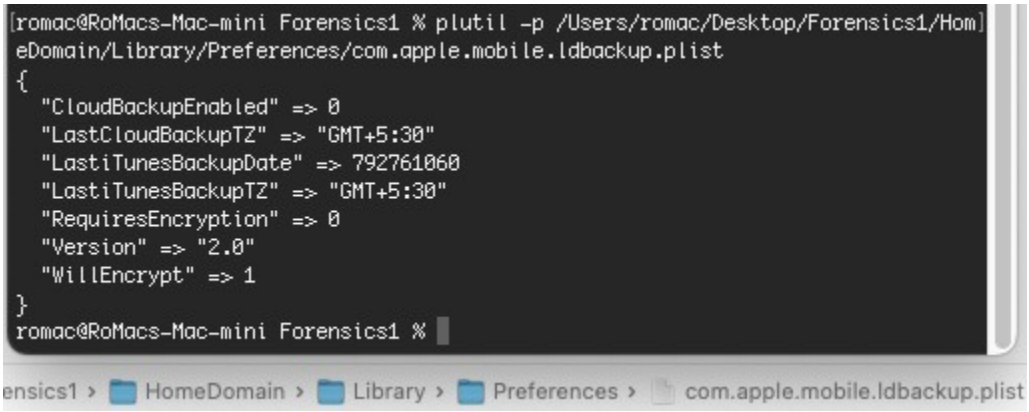


Figure 5.26: The com.apple.mobile.ldbackup.plist in plist Editor Pro

This records information such as the timestamps of the most recent local and iCloud backups, the time zone associated with the backup event, and whether backup encryption was enabled. Artifacts of this nature can help establish backup timelines, identify additional potential evidence sources, and corroborate acquisition history during an investigation.

Important .plist files

Filesystem extractions and device backups typically contain numerous plist files that may provide investigative value. These artifacts can reveal device configuration, account usage, network history, application behavior, and restoration activity. Because Apple frequently modifies storage paths and structures between iOS releases, you should treat file locations as version-dependent and validate findings against the acquisition source.

The following plist files are commonly encountered and may contain information relevant to forensic analysis:

plist file	Description
`/HomeDomain/Library/Preferences/com.apple.commcenter.shared.plist`	May contain the active phone number and telephony configuration details.

plist file	Description
`/HomeDomain/Library/Preferences/com.apple.identityservices.idstatuscache.plist`	Stores Apple ID–related identifiers and metadata associated with FaceTime and iMessage communications.
`/HomeDomain/Library/Preferences/com.apple.mobile.ldbackup.plist`	Records timestamps of recent local and iCloud backups, backup time zone, and whether encryption was enabled.
`/HomeDomain/Library/Preferences/com.apple.MobileBackup.DemotedApps.plist`	Lists applications manually or automatically offloaded by the system due to inactivity.
`/HomeDomain/Library/Preferences/com.apple.mobilephone.speeddial.plist`	Contains favorite contacts and speed dial associations.
`/HomeDomain/Library/Preferences/com.apple.preferences.datetime.plist`	Stores time zone configuration and related time settings.
`/RootDomain/Library/Caches/locationd/clients.plist`	Identifies applications that have requested or used location services.
`/RootDomain/Library/Preferences/com.apple.MobileBackup.plist`	Provides metadata related to device restore events, including restore timestamps, file transfer statistics, and source device identifiers.
`/private/var/containers/Shared/SystemGroup/systemgroup.com.apple.mobileslaltcache/Library/Caches/com.apple.MobileGestalt.plist`	Contains MobileGestalt device properties and may include Wi-Fi-related information useful for device identification and forensic analysis.
`/WirelessDomain/Library/Preferences/com.apple.commcenter.plist`	May contain carrier configuration details, ICCIDs, IMSIs, and additional cellular metadata.

Table 5.1: Common plist artifacts and their forensic relevance in iOS

Note

The location and availability of plist artifacts can change between iOS versions and acquisition types. Paths shown here should therefore be treated as common reference locations rather than guaranteed paths for every device or iOS release.

Although plist files typically do not contain the volume of user content found in SQLite databases, they often provide valuable contextual information that supports timeline reconstruction, account attribution, and device usage analysis. Correlating plist data with primary artifacts can significantly strengthen investigative conclusions.

Although SQLite databases and plist files represent the primary sources of structured forensic evidence, investigators should not overlook several additional artifact categories that may provide valuable context. These files often capture user behavior, media activity, personalization patterns, and application usage details that complement database-driven findings.

Local dictionary

iOS maintains a custom dictionary containing words manually added by the user through typing behavior and autocorrect learning. This artifact can provide insight into usernames, slang terms, contact nicknames, or frequently typed phrases that may not appear elsewhere in structured data.

In filesystem extractions, dictionary content can typically be found within keyboard-related directories under the user library domain. Because the data is often stored in a relatively simple format, it can be reviewed using standard text or plist viewers depending on the iOS version. While this artifact is subtle, it can be particularly useful in attribution scenarios or when reconstructing partially deleted communication content.

Photos and media files

User photographs and videos remain one of the most information-rich evidence sources on an iOS device. Media files captured using the device camera are commonly stored within the DCIM directory under the user media domain. This location may contain camera photos, screenshots, edited media, and cached derivatives generated by the Photos framework.

Each image may include exchangeable image file format (EXIF) metadata describing capture time, device model, orientation, and potentially geographic coordinates when location services were enabled. Tools such as ExifTool can be used to extract this metadata and assist with timeline reconstruction or location attribution. Investigators should also be aware that some third-party applications may save media within their own sandbox containers rather than the main camera roll, requiring examination of individual application directories.

Thumbnails and derived media

iOS generates thumbnails and preview images to support gallery browsing and system indexing. These artifacts may persist even after the original media has been deleted, making them valuable in recovery scenarios.

Thumbnail data can be found within Photos framework cache directories and related media metadata locations. Although earlier iOS versions relied heavily on ithmb files, modern iOS releases increasingly use alternative caching formats and database-driven thumbnail storage. Regardless of format, thumbnail remnants may still provide visual confirmation of deleted media content.

Wallpaper and personalization artifacts

Device personalization can also yield investigative clues. Wallpaper images, lock screen thumbnails, and SpringBoard configuration artifacts may reflect user identity, affiliations, or behavioral context.

Wallpaper thumbnails are typically stored within SpringBoard-related directories and may survive device activity changes longer than expected. While not always evidential on their own, these artifacts can support user attribution or device ownership assessments.

Third-party application data

In modern investigations, user activity is often concentrated within third-party applications rather than native system apps. Messaging platforms, social networking clients, cloud storage apps, and collaboration tools can collectively contain a significant portion of the user's communications, media exchanges, and behavioral history. As a result, forensic analysis must extend beyond built-in artifacts and account for application-specific storage patterns.

Each third-party application operates within its own sandbox container, typically located under `/private/var/mobile/Containers/Data/Application/`. These containers are identified by randomly generated UUID directory names rather than application titles. Investigators must therefore map container identifiers to application bundle names using metadata, container preference files, or forensic tooling that resolves these relationships automatically.

Within an application container, relevant artifacts are commonly distributed across the documents, library, and tmp directories. Persistent user data, including databases and media, is typically stored under Documents or Library, while caches and transient artifacts may appear under tmp. SQLite databases and property list files remain the most common storage formats, although some applications implement additional encryption or proprietary data serialization.

Messaging applications provide a useful illustration of this model. For example, WhatsApp maintains structured chat databases, media directories, and configuration files within its sandbox or shared app group container. Although messages are transmitted using end-to-end encryption, the device often retains locally stored message metadata and content that can be examined after acquisition, depending on the acquisition method and device state. Media exchanged through such applications is frequently stored separately from message databases, requiring correlation between structured records and filesystem artifacts to reconstruct conversations accurately.

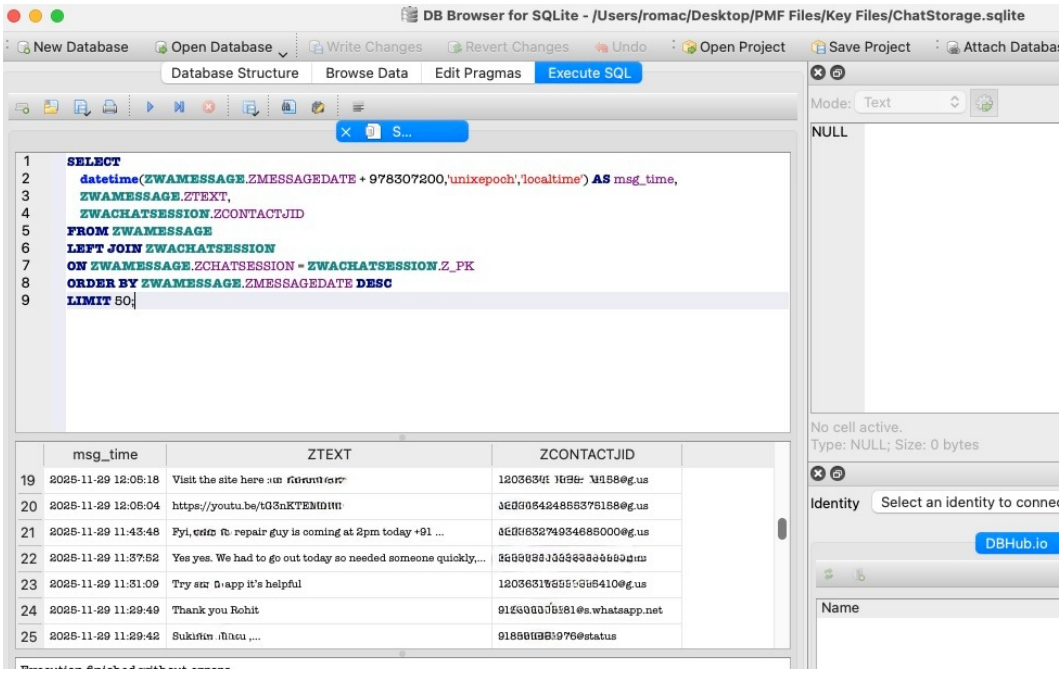


Figure 5.27: Querying WhatsApp messages from ChatStorage.sqlite using DB Browser for SQLite

Investigators should also be aware that third-party applications evolve rapidly. Storage locations, database schemas, and encryption mechanisms may change between application versions, and some apps implement additional protections that limit manual analysis. Furthermore, application data may synchronize with cloud services, creating secondary evidence sources beyond the physical device.

For these reasons, third-party application analysis requires a flexible approach that combines container identification, database examination, media correlation, and validation against acquisition scope. Even when automated tools parse these artifacts, understanding where the data originates and how it is structured remains essential for accurate interpretation and defensible reporting.

Applying artifact analysis beyond core databases

While this section examined several of the most commonly encountered iOS artifacts, it is important to recognize that no single chapter can comprehensively cover every database, file, or application-specific storage location present on modern devices. Apple continuously evolves data structures across iOS versions, and third-party applications introduce their own storage models, encryption methods, and artifact patterns.

The goal of this section was therefore to provide a practical methodology for identifying, locating, and interpreting artifacts regardless of where they reside. By understanding how iOS organizes data, how SQLite and plist structures are used, and how backups and filesystem acquisitions expose these artifacts, you can confidently extend the same analytical approach to previously unseen databases and applications.

In real investigations, artifact selection should always be driven by investigative objectives. Rather than searching for specific files in isolation, you should focus on the questions you are attempting to answer, such as communication patterns, user intent, behavioral timelines, and data persistence, and then apply the techniques learned in this chapter to uncover the most relevant sources of evidence.

Having examined how active artifacts are identified and interpreted, the next step is to consider what happens when relevant data has been deleted and whether any traces of it can still be recovered.

Deleted data recovery and limitations

One of the most misunderstood aspects of iOS forensics is the concept of **deleted data**. It is common to assume that when a user deletes a message, call entry, or photo, that data disappears immediately and permanently. In practice, deletion on iOS is usually a logical operation rather than a physical one.

The operating system marks storage space as available for reuse, but it does not instantly overwrite the underlying bytes. Whether deleted information can still be recovered depends on how the data was stored, how much time has passed, the version of iOS involved, and the method used to acquire the device. Understanding how deletion works at a structural level is essential for setting realistic expectations during analysis.

When a record is deleted from an SQLite database, it is removed from the active database structure, but portions of the underlying data may remain within database pages until that space is reused or overwritten. Depending on how SQLite manages the freed space, remnants may persist within freeblocks, freelist pages, or associated WAL files.

Because iOS relies heavily on SQLite databases to store application data, this behavior creates opportunities for forensic recovery. Messages, contacts, notes, call history, and many third-party application artifacts are stored in SQLite format. Let us now examine potential options and their relevance.

SQLite freelist pages

SQLite organizes data into fixed-size pages. When records are deleted, SQLite may leave residual data within unused portions of active pages, while pages that are no longer required may be added to the `freelist` for later reuse. From a forensic standpoint, both areas can potentially contain remnants of deleted records until the space is reused or overwritten. The freelist acts as a pool of reusable pages that SQLite can draw from when new data is inserted.

From a forensic standpoint, this is significant. Deleted records may still reside within these freelist pages until new data overwrites them. It is not uncommon to recover fragments or entire deleted entries for SMS messages, call records, contacts, notes, or metadata associated with attachments. Commercial forensic tools often include dedicated routines to scan freelist pages and reconstruct deleted records. Open-source tools and manual hex-level examination can also reveal these remnants, provided the database has not been aggressively compacted.

However, recovery from freelist pages is not guaranteed. If the application has been heavily used since deletion, or if the database has been vacuumed or compacted, the previously freed pages may have been overwritten. The more active the device, the shorter the recovery window typically becomes.

The following figure illustrates freelist pages as displayed in a commercial forensic tool, revealing message records that have been deleted:

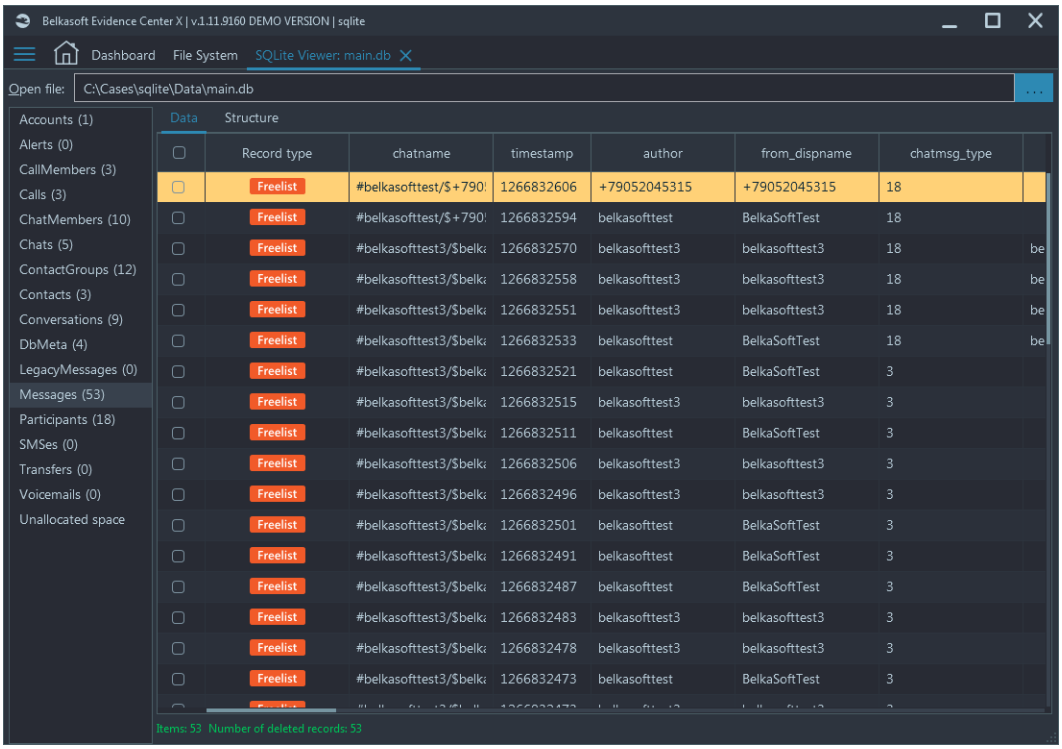


Figure 5.28: Examining freelist pages in BelkaSoft tool

Figure 5.28 shows records or fragments identified by the tool within SQLite freelist pages. A freelist page is no longer allocated to an active table or index, but its contents do not by themselves prove that a user deliberately deleted a corresponding record. The page may have become free through row deletion, table rebuilding, vacuuming behavior, application maintenance, schema migration, or other database operations.

You should validate the recovered fields against the database schema and corroborate them with active records, WAL frames, related tables, attachments, timestamps, or other evidence before drawing conclusions about deletion or user intent.

> **Note**
>
> It is important to note that these freelist pages may not be readily visible in standard SQLite viewers such as DB Browser for SQLite. Accessing and interpreting this data requires low-level parsing of the SQLite database structure rather than relying solely on standard database APIs.

Write-ahead log (WAL) files

Modern versions of iOS commonly use SQLite in **write-ahead log** mode. In this configuration, changes to a database are first written to a separate file known as the **WAL file** before being merged back into the main database. This design improves performance and reliability but also introduces forensic implications. A WAL file may contain recently inserted records, recently deleted entries, or transactions that were not yet merged into the main database. In some cases, records that are no longer visible in the primary database can still be found inside the associated WAL file. For this reason, WAL files often provide valuable insight into recent user activity.

The following figure shows records identified within the write-ahead log associated with the examined SQLite database. The entries marked Wal are individual records recovered from WAL pages; they do not represent separate `.wal` files. The left pane shows the database tables, while the main pane displays records and their associated fields recovered from the WAL. These records may include recently inserted, modified, or deleted data that has not yet been fully reflected in the main database.

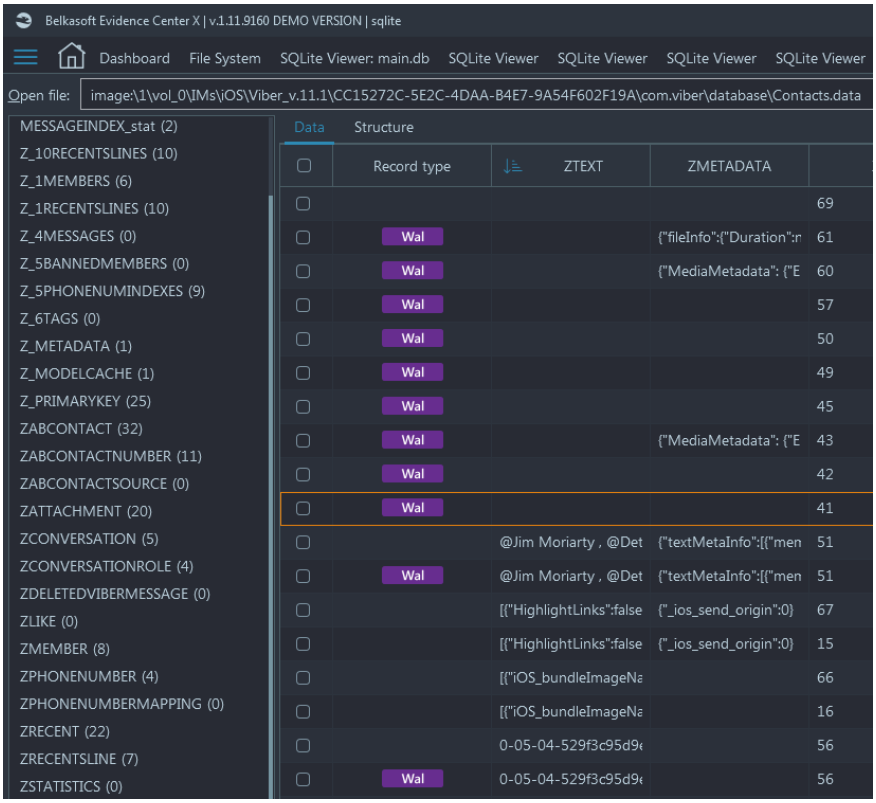


Figure 5.29: Examining records recovered from SQLite WAL pages using Belkasoft

However, WAL content is not permanent. SQLite periodically checkpoints WAL transactions back into the main database, after which the WAL may be truncated, reset, or reused. Application activity, database maintenance, and checkpoint operations can therefore reduce or eliminate recoverable historical WAL content. Backup-based extractions may also omit WAL and SHM files because iOS backups contain a selected set of files rather than a complete copy of the application filesystem. Filesystem-level acquisitions are more likely to preserve these companion files because they capture the application's directory structure more directly.

Tip

When examining an SQLite database, preserve and analyze its associated `-wal` and `-shm` files together with the main database. Opening or copying only the main database may result in recent transactions or other recoverable records being overlooked.

Unallocated space in file-based acquisitions

In traditional computer forensics, investigators often carve unallocated disk space in search of deleted files. On modern iOS devices, this approach is severely limited. Since the introduction of hardware-backed encryption and file-based encryption tied to the Secure Enclave, raw flash memory is encrypted at the hardware level. Even if a bit-by-bit image of storage were obtained, the data in unallocated space would typically remain encrypted and unreadable without the appropriate cryptographic keys.

As a result, physical carving of deleted files from raw storage is largely infeasible on contemporary iPhones. Deleted data recovery is now focused primarily on structured remnants within databases and application storage, rather than low-level disk carving. This represents a significant shift from legacy mobile forensics practices.

Recently deleted vs. soft-delete artifacts

Not all deletions are equal. Many iOS applications implement a **soft-delete** mechanism. Instead of permanently removing data, the application flags the item as deleted and moves it into a "recently deleted" state. Photos, notes, and, in newer versions of iOS, certain message categories follow this model.

In these cases, the data remains fully intact for a defined retention period or until the user explicitly purges it. From a forensic perspective, such artifacts are often straightforward to recover because they still exist in structured form within the application database or file system. You should always verify whether a deletion was permanent or merely logical within the application's lifecycle.

Impact of modern iOS security

While SQLite freelists and WAL files still provide recovery opportunities, modern iOS security significantly limits what is realistically recoverable. Key restrictions include the following:

- **Hardware-based encryption**: Each file is encrypted using keys tied to the device's Secure Enclave and passcode. Without the device being unlocked at the time of acquisition, meaningful low-level recovery becomes nearly impossible.
- **Aggressive database cleanup**: Newer iOS versions increasingly optimize storage and compact databases. Vacuum operations or database maintenance can permanently remove deleted records.
- **Cloud synchronization**: If a user deletes data on one device and it syncs via iCloud, deletion may propagate quickly across devices and backups.

- **Application-level encryption**: Some third-party applications encrypt their own SQLite databases or message stores, preventing manual analysis even when remnants are present.
- **Advanced data protection**: With end-to-end encryption extended to more iCloud categories, some deleted data may never be accessible through cloud channels without the user's private keys.

In short, while deleted data recovery remains possible, it is far more constrained than in earlier generations of mobile devices.

Practical expectations for examiners

In contemporary iOS investigations, deleted data recovery should be approached with measured expectations. It is realistic to recover deleted records from SQLite freelists or WAL files when the acquisition method captures those structures and when the data has not yet been overwritten. Soft-deleted content within applications can also yield useful artifacts.

What is far less realistic is the recovery of arbitrarily deleted files from raw storage on modern, encrypted devices. Without proper credentials or decrypted access, physical carving offers little value. Similarly, strongly encrypted third-party application data may remain inaccessible without the necessary keys.

Deleted data recovery today is less about deep disk carving and more about intelligent database analysis, careful correlation, and a clear understanding of platform limitations. Your role is not merely to extract data, but to interpret what remains and to explain, with technical accuracy, why certain artifacts can or cannot be recovered.

Summary

This chapter focused on the phase that follows acquisition: interpreting and validating iOS data to uncover meaningful evidence. Although forensic tools can automatically parse large volumes of artifacts, dependable analysis requires you to understand how iOS stores data and to verify tool output against raw evidence when necessary. We began by examining timestamp interpretation, including Unix time, Mac Absolute Time, and application-specific formats. Correct conversion is essential for constructing defensible timelines and correlating activity across artifacts.

The chapter then explored SQLite databases as the primary storage mechanism for structured iOS data. Understanding database structure, table relationships, and query methods enables you to validate findings, recover overlooked artifacts, and supplement automated parsing. Techniques for locating SQLite databases in both filesystem extractions and hashed backup structures were also discussed. Key forensic artifacts such as contacts, call history, messages, notes, Safari activity, device interaction data, and cellular usage were analyzed to demonstrate

how user behavior and communication patterns can be reconstructed. The role of property list files and other configuration artifacts in revealing device state and system behavior was also highlighted.

Finally, we addressed deleted data recovery and its limitations. While remnants may persist in freelists, WAL files, and soft deletion states, modern iOS encryption, database optimization, and cloud synchronization significantly reduce deep recovery opportunities. As a result, effective deleted data analysis increasingly relies on structured database examinations and artifact correlation. By understanding artifact locations, structure, and validation methods, you can move beyond tool-driven analysis and adopt a defensible investigative approach.

The next chapter builds on this foundation by introducing forensic tools and automation techniques that help streamline analysis while maintaining evidentiary integrity.

6

iOS Forensic Tools and Automation

In the previous chapters, we examined how to acquire data from iOS devices and how to manually interpret core artifacts such as messages, call logs, Safari history, Notes databases, and system intelligence records. While understanding artifact structure is essential, modern iOS investigations rarely involve examining only a handful of databases. Today's devices contain tens of thousands of files, multiple gigabytes of structured data, encrypted containers, synchronized cloud artifacts, and constantly evolving third-party application schemas.

On this scale, manual analysis alone is neither practical nor efficient. **Forensic tools** and **automation** help you process large datasets, identify relevant artifacts, reconstruct activity, and automate repetitive analysis tasks. However, automation must be used carefully. Parsing errors, unsupported schemas, or incorrect assumptions can propagate across large datasets and undermine the reliability of the resulting evidence.

This chapter examines how commercial forensic platforms, open-source utilities, and automation techniques can be used together to improve the efficiency and consistency of iOS forensic analysis while preserving the validation and evidentiary discipline required for forensic investigations.

In this chapter, we will cover the following key topics:

- iOS forensic tooling landscape
- Tool validation and evidentiary reliability
- Role and limits of automation in iOS forensics
- Working with commercial iOS forensic platforms
- Automation techniques for iOS artifact analysis

Note

The examples in this chapter generally target iOS 26 unless otherwise stated. Some screenshots use datasets acquired from earlier iOS versions where the demonstrated artifact or workflow remains applicable. Our aim is to demonstrate forensic analysis and automation techniques rather than to establish compatibility with a specific iOS release.

iOS forensics tooling landscape

Modern iOS investigations can involve large and complex datasets. A contemporary iPhone may contain hundreds of applications, extensive media collections, messaging databases, cloud synchronization artifacts, system-generated activity records, and data distributed across multiple applications and system containers.

Filesystem acquisitions expose much of this information within the original iOS directory structure. Third-party application data, for example, is commonly stored within sandboxed containers under `/private/var/mobile/Containers/Data/Application/`. These containers are identified by UUIDs rather than human-readable application names and may contain SQLite databases, write-ahead log (WAL) files, property lists, caches, media, and other application data. Determining which container belongs to a particular application may itself require correlation with application metadata or bundle identifiers.

Backup-based evidence introduces a different challenge. iOS backups are created through Finder or iTunes and store files using SHA-1-derived identifiers rather than their original filenames. As discussed in *Chapter 5*, you can use `Manifest.db` to map these identifiers back to their original domains and relative paths before examining individual artifacts.

On this scale, manually locating and interpreting every potentially relevant artifact is rarely practical. Forensic tools help reduce this complexity by acquiring, identifying, parsing, organizing, and presenting evidence for examination. However, not every forensic tool performs the same role.

Functional roles of iOS forensic tools

Across the forensic workflow, tools generally support one or more of three functional areas:

- **Acquisition**: These capabilities focus on obtaining data from iOS devices, backups, or cloud sources. Depending on the tool and acquisition method, this may involve device communication, authentication, backup processing, filesystem extraction, or handling encrypted evidence.

- **Processing and parsing**: Once data has been acquired, parsing engines identify known artifacts and interpret their underlying structures. This may involve reading SQLite databases and property lists, resolving relationships between database tables, converting timestamps, interpreting application-specific structures, and examining associated WAL or residual database content.
- **Analysis and presentation**: Parsed information is organized into investigator-friendly views such as conversations, call histories, browser activity, media collections, and timelines. Search, filtering, correlation, bookmarking, and reporting capabilities help you move from a large dataset toward evidence relevant to the investigation.

These functions do not necessarily exist within a single product. Some tools specialize in device access and acquisition, while others focus primarily on processing, analysis, or reporting. You may therefore use one tool to acquire evidence and another to analyze the resulting extraction.

Understanding these functional areas is also important when evaluating forensic results. Incomplete evidence may originate from limitations in the acquisition itself, while an incorrect or missing artifact may result from parsing logic or schema support. The final information displayed to you may therefore have passed through several stages of processing before appearing in a report. For this reason, parsed tool output should be treated as an interpretation of the underlying evidence rather than the evidence itself.

Commercial forensic platforms

Commercial forensic platforms are widely used because they can process large datasets and automate many common investigative tasks. Platforms such as Cellebrite, Magnet AXIOM, Oxygen Forensic Detective, and Belkasoft provide capabilities across different stages of the mobile forensic workflow.

Depending on the platform, these tools can identify known artifacts, reconstruct conversations, interpret application databases, normalize timestamps, correlate related records, build timelines, search across multiple evidence sources, and generate structured reports. Some platforms also provide capabilities for identifying residual or deleted records within SQLite structures such as freelist pages and WAL files.

The primary advantage of commercial forensic platforms is scale. Instead of manually locating and examining hundreds of databases and configuration files, you can rapidly identify commonly supported artifacts and focus attention on data relevant to the investigation. This becomes particularly valuable when examining multiple devices, large filesystem extractions, backups, or evidence from several sources.

However, automated parsing depends on the artifact definitions and interpretation logic implemented by the tool. Apple frequently modifies system databases, and third-party

applications can change their schemas, storage locations, or data formats between versions. A parser designed for an earlier structure may therefore produce incomplete results or fail to recognize a newer artifact until support is updated.

Commercial platforms may also abstract much of the underlying parsing logic from the examiner. A tool may display a message, timestamp, location, or application event without exposing every query or transformation used to derive that result. This does not make the finding unreliable, but significant findings should be traceable to and, where necessary, validated against the underlying evidence.

Open-source and manual analysis tools

Manual and open-source tools complement commercial platforms by allowing investigators to work directly with the underlying artifacts. Utilities such as sqlite3, DB Browser for SQLite, and property list inspection tools can be used to examine database schemas, query records, inspect raw values, and verify how information is actually stored. This visibility is particularly useful during validation. If a commercial platform reports a significant message, timestamp, or application event, you can inspect the source database to confirm that the corresponding record exists and determine how the reported value was derived.

Manual analysis is also valuable when automated parsers do not support an artifact. Newly released applications, changed database schemas, uncommon applications, or partially corrupted data may require direct examination before commercial parser support becomes available. In such cases, understanding SQLite, property lists, application containers, and other underlying iOS data structures allows the investigation to continue without depending entirely on predefined artifact support.

Manual tools, however, do not provide the same level of automated discovery, correlation, and reporting as comprehensive forensic platforms. Examining every artifact manually would quickly become inefficient for a large acquisition. Their greatest value therefore lies in targeted analysis, validation, and examination of unsupported or unusual artifacts.

Combining automated and manual analysis

In practice, effective iOS forensic analysis often combines both approaches. Commercial platforms can be used to process and triage large datasets, identify known artifacts, and surface potentially relevant evidence. Manual and open-source tools can then be used to investigate unsupported artifacts, resolve inconsistencies, and validate findings that are important to the investigation.

The choice of tool should therefore be driven by investigative tasks rather than by an assumption that a single platform can provide every answer. A tool that performs acquisition particularly well may not provide the deepest artifact analysis, while another may offer stronger parsing or correlation capabilities. Using multiple approaches also provides an

important safeguard against relying on a single interpretation of the evidence. When a finding is significant, you should be able to trace it back to its source and understand enough of the underlying artifact to explain how the conclusion was reached.

Tip

Do not evaluate a forensic workflow solely by how many artifacts a tool reports. Different platforms may acquire, classify, parse, and count artifacts differently. The objective is not to produce the largest number of results, but to identify evidence that is relevant, accurate, and defensible.

This combination of automated processing and examiner-driven validation provides the foundation for the workflows discussed throughout this chapter. In the following section, we will examine the reliability of forensic tool output in greater detail and the steps you can take to validate significant findings.

Tool validation and evidentiary reliability

Forensic tools accelerate analysis, but speed does not equate to accuracy. In digital investigations, particularly those involving encrypted mobile ecosystems such as iOS, the credibility of evidence depends not only on what is found, but on how it is validated. Courts evaluate methodology as closely as conclusions.

An examiner who cannot explain how a record was derived, interpreted, and verified risks undermining the evidentiary value of otherwise relevant data. **Tool validation** is therefore not an optional technical exercise; it is central to professional forensic practice. Recent comparative research reinforces this point. Studies comparing major mobile forensic platforms have shown that the same device can produce significantly different artifact counts across tools. Such differences do not necessarily mean that one tool recovered more evidential data; they may also result from differences in artifact classification, parsing logic, and provenance. For this reason, artifact counts alone should never be used as a measure of forensic completeness.

Cross-tool validation

When a finding is particularly significant, processing the same evidence with a second forensic tool can provide an additional form of validation. Agreement between independent parsers can increase confidence in the result, while disagreement can identify artifacts that require closer manual examination.

Differences between tools should not automatically be interpreted as evidence that one tool has failed. Forensic platforms may use different parsing logic, support different database schemas, classify artifacts differently, or interpret the same underlying records in different

ways. As a result, two tools may report different artifact counts or representations while processing the same evidence.

When discrepancies occur, as the examiner, you should return to the underlying artifact and determine which interpretation is supported by the source data. **Cross-tool comparison** is therefore most useful to identify findings that require validation rather than as a simple measure of which tool recovered more evidence.

Source artifacts vs. parsed output

A fundamental principle in mobile forensics is the distinction between source artifacts and parsed output. Source artifacts include the files preserved within the forensic acquisition, such as SQLite databases, WAL files, property lists, logs, and other structured system files. **Parsed output**, by contrast, is a representation generated by a forensic tool's interpretation engine and is typically produced through automated queries, schema mappings, and timestamp conversions.

When a forensic tool displays a message thread, call log, or browser history entry, it is not presenting the raw database content verbatim. Instead, it is presenting an interpreted reconstruction derived from structured data stored within the underlying artifacts. This reconstruction typically involves multiple processing steps. A tool may join several related tables to reconstruct a conversation thread, convert internal timestamp values into human-readable dates, resolve foreign key relationships between database records, interpret encoded or compressed content fields, and filter records based on internal parsing logic.

Because this interpretation layer sits between the examiner and the underlying data, errors can occur even when the database itself remains intact. A parser may misinterpret a column, apply an incorrect timestamp conversion, or join tables improperly. For this reason, you must always be able to trace a reported artifact back to the original source file and confirm that the record exists within the primary evidence. A simple but powerful validation principle is therefore to ask the question: *Where in the primary evidence does this record reside?*

Timestamp verification and chronological integrity

Chronological reconstruction often plays a central role in digital investigations. Forensic platforms typically convert raw **timestamps** into readable dates automatically, but these converted values should not always be accepted without verification. As discussed in *Chapter 5*, iOS artifacts may use different timestamp formats, units, and time representations depending on the database and application.

For significant findings, you should sample raw timestamp values directly from the source artifact and independently reproduce the tool's conversion. The independently derived time can then be compared with the value presented by the forensic platform. This helps identify

problems such as incorrect epoch assumptions, unit interpretation, time zone conversion, or loss of precision.

Verification should also consider consistency across related artifacts. For example, if a message timestamp is important to an investigative timeline, you can compare the parsed timestamp with the raw value in the underlying database and correlate it with related records such as attachments, notifications, or other system activity. A discrepancy should be investigated rather than automatically attributed to either the forensic tool or the source data.

Tip

When documenting a significant timestamp, retain both the original raw value and the normalized human-readable value. This makes the conversion reproducible and allows another examiner to independently verify the result.

Schema interpretation and record count confirmation

As discussed in *Chapter 5*, iOS artifacts frequently use SQLite databases in which relevant information may be distributed across multiple related tables. When validating forensic tool output, the objective is not to reconstruct these schemas from first principles, but to confirm that the tool has interpreted the relevant records and relationships correctly.

A practical validation method is to compare the number of records reported by the forensic tool with the underlying database. For example, if a tool reports a specific number of messages in a conversation, you can independently query the relevant tables and compare the resulting count. Differences do not necessarily indicate an error. A parser may exclude system-generated records, duplicates, deleted entries, or unsupported record types. Any significant discrepancy should therefore be understood and documented rather than assuming either result is correct.

Join logic should also be validated when it affects an important finding. For example, if a tool associates a message with a particular contact by joining the message and handle tables, you can reproduce the relevant join against the source database and confirm that the relationship matches the tool output. This is particularly important after iOS or application updates, when schema changes may cause an existing parser to interpret relationships incorrectly or omit newly introduced fields.

Deleted data claims and residual artifact validation

Deleted data recovery remains one of the most closely scrutinized aspects of mobile forensics. Many forensic platforms attempt to identify residual records within SQLite freelist pages or within WAL files. While such recovery is technically possible, it must be approached carefully.

When a tool reports recovered deleted records, you should confirm that the record is no longer present in the active database tables and that the corresponding data exists within freelist pages or WAL content. The recovered record should also be examined to ensure that it remains structurally coherent and has not been partially overwritten by subsequent data.

Residual database pages may contain fragments of older records, and forensic tools must distinguish between intact records and corrupted remnants. Overstating deleted data recovery without validation can lead to the presentation of artifacts that are incomplete or misinterpreted. In high-stakes cases, manual inspection of database pages or WAL content may be necessary to substantiate recovery claims. Such validation demonstrates that the evidence was not merely inferred by a tool but directly observed within the primary artifact.

Version compatibility and parser limitations

The iOS platform evolves continuously. Each major release may introduce schema changes, modify storage locations, adjust encryption handling, or restructure application data models. Third-party applications also update frequently, sometimes altering database structures without notice.

Forensic tools must adapt to these changes, but adaptation is rarely immediate. A parser that performs accurately on one iOS version may behave differently on another. For this reason, you should document the iOS version of the device under examination as well as the version of the forensic tool used during analysis. Any observed parser limitations or inconsistencies should also be noted.

Testing tools against controlled datasets can reveal how they handle particular artifacts and help establish **repeatability**. Repeatability or the ability for another qualified examiner to reproduce findings using the same dataset and methodology, is a key component of evidentiary credibility. If results cannot be reproduced consistently, their reliability may be questioned.

Documentation and defensibility

Validation activities should be documented alongside the findings they support. When a significant artifact is independently verified, you should record the forensic tool and version used, the source artifact examined, any manual queries or conversion methods applied, and any discrepancies identified between the parsed output and the underlying data. This information establishes how a particular tool-generated result was verified and allows another examiner to reproduce the validation if necessary. Where different tools produce different results, the discrepancy and the method used to resolve it should also be documented rather than simply reporting the preferred result.

Maintaining this validation record is particularly important when parser behavior may depend on the iOS version, application version, database schema, or tool version. It provides a clear

link between the automated result, the underlying evidence, and the examiner's independent verification.

These validation principles establish how tool-generated findings can be tested and supported. The next section considers a related question of which parts of iOS forensic analysis should be automated in the first place and which decisions should remain with the examiner.

Role and limits of automation in iOS forensics

Automation has become an important part of modern iOS forensic analysis. A single acquisition may contain thousands of files, databases, property lists, application containers, and system artifacts. Manually examining every item is neither practical nor necessary. Forensic tools and custom scripts can reduce this workload by identifying known artifacts, extracting relevant fields, organizing results, and highlighting information that warrants further examination.

However, automation should be viewed as a way to accelerate analysis rather than replace investigative judgment. An automated process operates according to predefined rules. It can identify a database, execute a query, convert a timestamp, or classify a record, but it does not inherently understand the investigative significance of what it finds.

This distinction becomes important when artifacts are ambiguous. A record may indicate that an application was active, for example, but this does not necessarily establish that the user intentionally interacted with it at that moment. Similarly, the presence of contact, location, or URL on a device does not by itself explain how it arrived there or what role it played in the investigation. Those conclusions require interpretation and, often, correlation with additional evidence.

What should be automated?

The best candidates for automation are repetitive and deterministic tasks where the same operation must be performed consistently across large amounts of evidence. Examples include identifying known artifact files, extracting predefined database fields, locating records that match particular criteria, converting data into a common representation, and organizing results for subsequent analysis.

Automation is particularly valuable during initial processing and triage. Instead of manually opening hundreds of databases and configuration files, an automated workflow can identify potentially relevant artifacts and direct you toward areas that deserve closer attention. The objective is not to automate every investigative step. Rather, automation should remove repetitive work so that the examiner can spend more time interpreting evidence, resolving inconsistencies, and answering investigative questions.

What should remain examiner-driven?

Interpretation should remain primarily examiner-driven. Automated tools can report that two events occurred within a short period, but deciding whether those events are related requires additional evidence. A parser can label a record as deleted, but you may need to determine whether the record represents an actual user deletion, a database state change, or residual data from an earlier transaction.

The same principle applies when automated results conflict. Different forensic tools may interpret the same underlying artifact differently because they use different parsing rules, schema assumptions, or classification logic. You should therefore be prepared to return to the original database or file and determine which interpretation is supported by the underlying evidence. Automation is most reliable when the question being asked is precise. As the required conclusion becomes more interpretive, the need for human analysis increases.

Automation can scale errors too

One of the risks of automation is that an incorrect assumption can be applied consistently across an entire dataset. A timestamp conversion that uses the wrong epoch or unit, for example, may produce thousands of apparently valid but incorrect dates. A SQL query written for an earlier database schema may omit records after an iOS update. A script may continue processing despite encountering an unexpected field and produce incomplete results without making the failure obvious.

Manual analysis usually limits an error to the artifact being examined while automation can reproduce the same error across hundreds or thousands of artifacts. For this reason, automated output should be treated in the same way as output from a commercial forensic tool: as an interpretation that must be validated when it contributes to an important investigative conclusion. Testing automation against known data, reviewing unexpected results, and periodically comparing automated output with the underlying artifacts help reduce this risk.

Maintaining provenance during automation

Automation should never separate a finding from its source. Extracted records should retain sufficient information to trace them back to the original evidence, such as the source acquisition, file path, database and table, and relevant record identifier. Where data is transformed, the original value should also be retained whenever practical.

This is particularly important when automated processing combines information from multiple databases or devices. A consolidated timeline may be useful for identifying patterns, but you must still be able to determine where each event originated and reproduce the process that produced it. Scripts, queries, configuration settings, and tool versions used during

automated analysis should therefore be documented as part of the examination. This allows significant findings to be reproduced and independently verified.

Human-guided automation

The most effective forensic workflows combine automation with examiner oversight. Automation performs repetitive processing and helps surface potentially relevant evidence, while the examiner determines what the evidence means, validates significant findings, and resolves ambiguity.

A useful principle is to automate collection and transformation but be cautious about automating interpretation and conclusions. The closer an automated decision comes to answering the ultimate investigative question, the more important human review becomes. This balance allows you to benefit from the speed and consistency of automation without losing the validation, context, and professional judgment required for defensible forensic analysis.

With these principles in mind, the next section examines how commercial iOS forensic platforms apply many of these capabilities in practice, from importing and processing acquired evidence to searching, correlating, validating, and reporting forensic artifacts.

Working with commercial iOS forensic platforms

Modern iOS investigations frequently rely on specialized forensic platforms to process and analyze extracted device data. While earlier sections discussed the conceptual role of forensic tools and the importance of validation, this section focuses on how these platforms are actually used during investigations. Commercial forensic suites provide integrated environments that combine artifact parsing, search capabilities, timeline construction, and reporting within a single interface.

Several forensic platforms are widely used in professional mobile forensic investigations. Examples discussed in this chapter include Cellebrite Inseyets (including Physical Analyzer), Magnet AXIOM, Oxygen Forensic Detective, Belkasoft Evidence Center, and Elcomsoft Phone Viewer. Although these platforms differ in their capabilities and areas of emphasis, they share many common analytical workflows for processing, searching, correlating, validating, and reporting iOS evidence.

Importing and preparing evidence

The first stage of analysis involves importing the acquired evidence into the forensic platform. Evidence sources may include logical extractions, filesystem acquisitions, encrypted backups, or datasets collected from cloud services. Once imported, the platform indexes the evidence and begins identifying the device details and known artifact structures.

Figure 6.1 shows an example of selecting an iCloud backup as an evidence source in Magnet AXIOM Process before the dataset is processed for analysis:

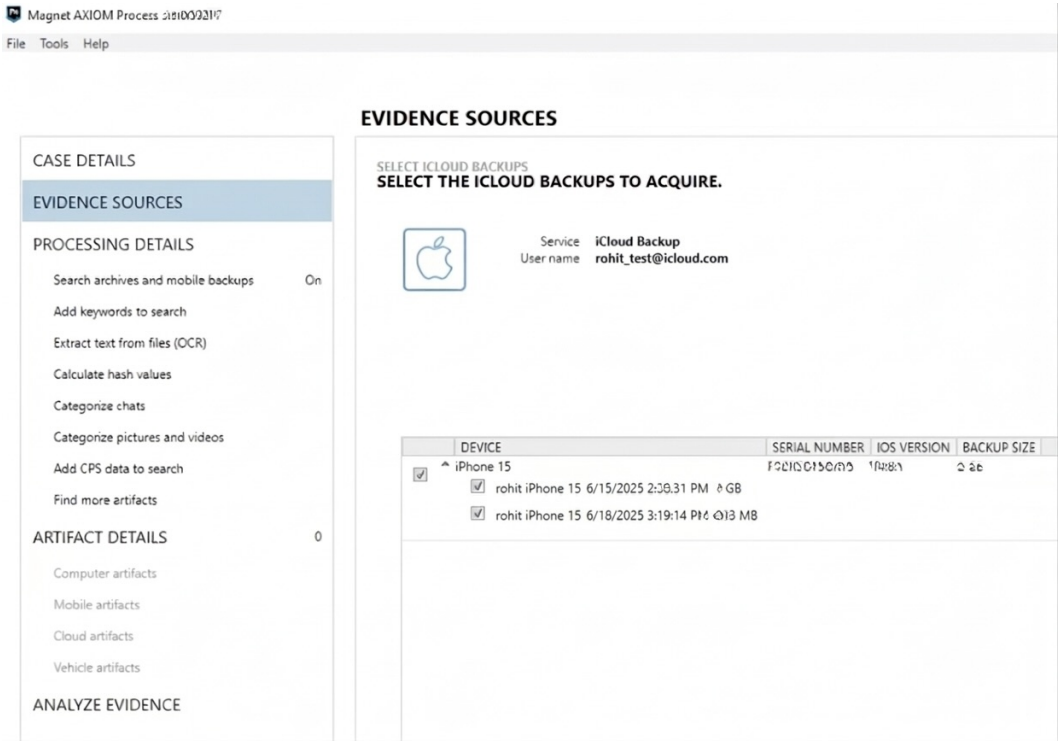


Figure 6.1: Selecting an iCloud backup as an evidence source in Magnet AXIOM Process

For example, when an encrypted iOS backup is loaded into tools such as Magnet AXIOM or Elcomsoft Phone Viewer, the software scans the dataset for known iOS artifact locations such as message databases, call history records, Safari browsing artifacts, and application containers. In the case of filesystem extractions processed in platforms such as Cellebrite Physical Analyzer or Oxygen Forensic Detective, the tool parses the entire directory structure and identifies databases located within system domains and application containers.

During this stage the software may also perform indexing operations to accelerate later searches and generate cryptographic hashes to ensure the integrity of the imported evidence. Most forensic platforms organize evidence within a case management structure. For instance, you may load several historical backups from the same device, enabling comparisons between different points in time, allowing you to associate multiple evidence sources with a single investigation.

Automated artifact parsing

After the evidence is ingested and indexed, the forensic platform begins parsing the dataset. **Parsing** involves interpreting structured artifacts from databases and configuration files and converting that information into human-readable records. For example, message databases such as `sms.db` may be parsed into threaded conversations, while call history databases such as `CallHistory.storedata` are converted into chronological call logs. Browsing artifacts may be reconstructed into lists of visited websites with associated timestamps and metadata.

Tools such as Cellebrite Physical Analyzer and Magnet AXIOM automatically identify many common iOS artifacts and present them within categorized views such as communications, internet activity, media files, and application usage.

Figure 6.2 shows an example of parsed call log records presented in Cellebrite Physical Analyzer:

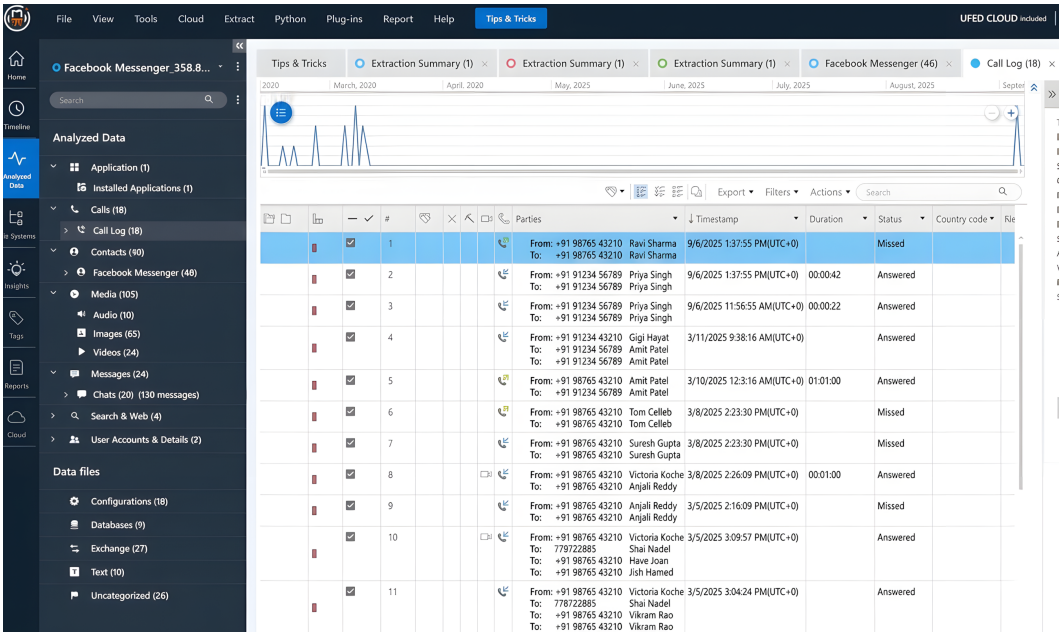


Figure 6.2: Parsed call log artifacts displayed in Cellebrite Physical Analyzer

Although automated parsing significantly accelerates analysis, you should remember that these categories represent the tool's interpretation of the underlying data. Parser definitions depend on knowledge of database schemas, and when Apple modifies internal structures or when applications change their storage formats, certain artifacts may not be parsed correctly. As a result, you should treat parsed output as an analytical starting point rather than a conclusion.

Searching and filtering artifacts

Once artifacts are parsed and categorized, you will begin the process of **searching and filtering** the dataset. Modern forensic tools provide powerful search capabilities that allow you to quickly identify relevant evidence within large collections of artifacts. Keyword searches can be applied across messages, notes, contacts, and other textual artifacts. You may also filter records by date ranges, communication participants, or application identifiers.

These filtering mechanisms allow you to reduce large volumes of data into manageable subsets aligned with investigative objectives. Effective filtering also allows you to quickly isolate communications or user activity occurring within a specific timeframe. This capability becomes particularly valuable when you are attempting to reconstruct events surrounding a particular incident.

Timeline analysis and activity reconstruction

One of the most valuable features provided by modern forensic platforms is **timeline analysis**. Timeline views combine artifacts from multiple sources and display them in chronological order. This allows you to reconstruct sequences of activity and identify relationships between events that might otherwise appear unrelated. For example, a timeline may reveal that a user visited a specific website shortly before sending a message or placing a call. It may also show that a particular application was active while communication events occurred.

Figure 6.3 shows how Oxygen Forensic Detective presents artifacts from multiple sources within a unified chronological timeline:

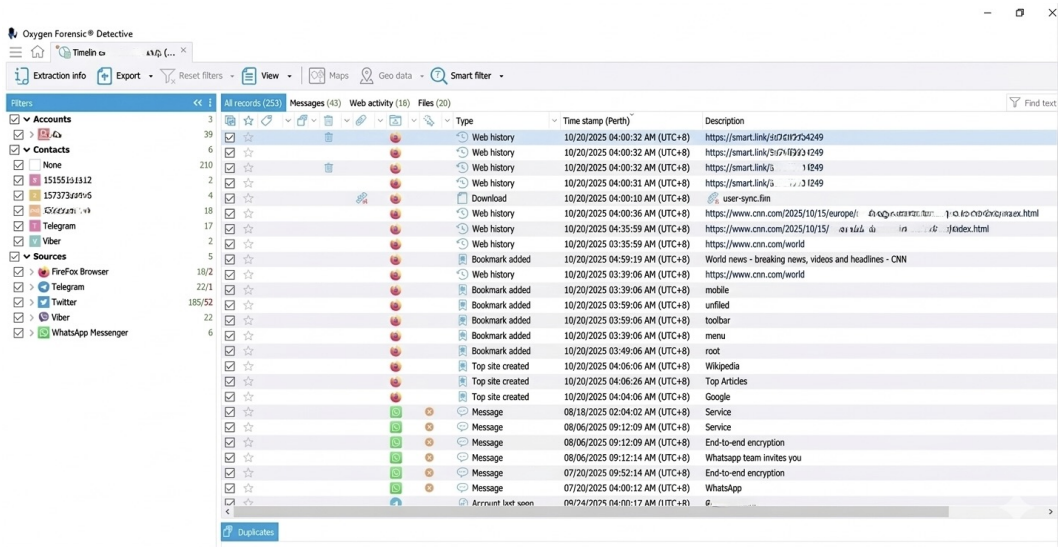


Figure 6.3: Timeline view in Oxygen Forensic Detective showing artifacts from multiple sources in chronological order

Timeline analysis becomes particularly useful when you must reconstruct user behavior during a specific time window. By examining events in chronological order, you can identify patterns and sequences that may help explain how a particular action occurred.

Validating critical findings

Before exporting findings, critical artifacts should be traced back to their underlying source data. A parsed result displayed by a forensic platform is an interpretation of the evidence, and significant findings should therefore be independently verified where practicable.

Validation may involve opening the underlying SQLite database or plist file, confirming the original timestamp and its conversion, examining associated WAL or other companion files, or comparing the artifact against another forensic tool or parser. The level of validation should reflect the significance of the finding, particularly when an artifact directly supports an important investigative conclusion.

Any differences identified during validation should be documented rather than automatically treated as tool errors. Differences may result from parser logic, database schema changes, artifact classification, or the way individual tools interpret the underlying data.

Exporting evidence and generating reports

The final stage of analysis involves **exporting relevant artifacts** and **generating reports** that document investigative findings. Forensic tools typically allow you to export individual

artifacts, selected records, or entire categories of parsed data. Reports may include message conversations, call logs, browsing history, media files, and associated metadata such as timestamps and identifiers. Many tools also provide options to include screenshots or previews of the original artifact views.

Clear reporting is essential because forensic findings must often be communicated to investigators, legal teams, or courts. Reports should therefore present evidence in a format that is understandable while still preserving the technical context necessary for validation.

How commercial platforms differ in practice

Although commercial forensic platforms support broadly similar investigative workflows, their capabilities and emphasis differ. Cellebrite Physical Analyzer, Magnet AXIOM, Oxygen Forensic Detective, Belkasoft Evidence Center, and Elcomsoft Phone Viewer may differ in the artifact types they recognize, how they present parsed and recovered records, their search and correlation capabilities, and the level of access they provide to underlying source data.

These differences are important during analysis because the same extraction may not be represented identically across platforms. One tool may recognize an application artifact that another does not, while another may provide stronger timeline, SQLite recovery, or cross-artifact correlation capabilities. Differences in artifact counts or classifications should therefore be investigated rather than assuming that one result is necessarily incorrect.

Note

The purpose of this section is not to provide product-specific interface walkthroughs, which can change between software releases, but to demonstrate the common analytical workflow used across commercial platforms and highlight where their behavior should be independently validated.

Commercial platforms provide the scale needed to process and triage modern iOS datasets, but their output remains most reliable when significant findings can be traced back to the underlying artifacts. The next section builds on this principle by examining how you can create your own automated workflows for repetitive artifact extraction, processing, and correlation.

Automation techniques for iOS artifact analysis

As discussed earlier in this chapter, modern iOS investigations frequently involve large volumes of data distributed across backups, filesystem extractions, and application containers. Even a single iPhone backup may contain thousands of files and dozens of SQLite databases representing different artifacts. When multiple backups or devices are involved, manually examining each artifact becomes increasingly inefficient.

Automation helps you to manage this complexity by standardizing repetitive analytical tasks. Instead of manually inspecting each database or performing the same queries repeatedly, you can use scripts and automated workflows to extract and process artifacts in a consistent manner. When implemented correctly, automation reduces analysis time while also improving consistency and reproducibility. This section explores several practical ways automation can be applied during iOS forensic analysis. This is not an exhaustive list but will give you an idea of what kinds of tasks can be automated.

Automating SQLite artifact extraction

Many key iOS artifacts are stored in **SQLite** databases. Messages, call history, browsing activity, contacts, and application data are typically stored in structured tables within these databases. Once you have identified the relevant tables and validated the correct query for extracting a particular artifact, that query can often be reused across multiple datasets. Instead of manually opening each database in a graphical viewer, you can use scripting languages such as Python to automate database queries. A simple script can open a database, execute a predefined query, and export the results to a structured format such as CSV.

For example, the following Python script extracts basic message data from the `sms.db` database:

```
import sqlite3

query = """
SELECT
    datetime(date/1000000000 + 978307200, 'unixepoch') AS message_time,
    text
FROM message
ORDER BY date DESC
"""

with sqlite3.connect("file:sms.db?mode=ro", uri=True) as conn:
    cursor = conn.cursor()

    for row in cursor.execute(query):
        print(row)
```

Running the script produces the extracted message records as timestamp and message-text pairs, as shown in the following figure:



Figure 6.4: Message timestamps and text extracted from sms.db using the Python script

The database is opened in read-only mode to prevent the script from intentionally modifying the evidence copy, and the context manager ensures that the database connection is closed when processing is complete. This example assumes that the date field uses nanoseconds relative to Apple's 2001 epoch. Before reusing the query against another iOS version or dataset, verify the timestamp representation and database schema. This approach allows you to quickly extract message records without manually browsing through database tables. Similar scripts can be written for other common artifacts such as call history, browser activity, or application logs.

Automating artifact extraction across multiple backups

In many investigations, you encounter more than a single iOS backup. A device may have been backed up repeatedly over time, resulting in multiple backup states or backup directories on a workstation. A backup reflects the data selected and successfully captured by the backup service; it is not a complete point-in-time representation of the device. Each backup represents the device state at a particular moment and may contain artifacts that no longer exist on the current device. For this reason, examining multiple backups can be extremely valuable when reconstructing timelines or identifying deleted communications.

However, analyzing several backups manually can quickly become inefficient. Each backup contains thousands of files renamed using SHA1 hashes, and locating a specific artifact such as the SMS database requires parsing the `Manifest.db` file for each backup individually. When dozens of backups are present, repeating this process manually becomes time-consuming and error-prone.

Automation provides a practical solution to this problem. By writing a simple script, you can automatically iterate through multiple backup directories, parse the `Manifest.db` database within each backup, and identify the location of specific artifacts. This approach allows you to rapidly identify relevant databases without manually navigating each backup structure.

Note

Automated workflows should preserve provenance alongside extracted or identified artifacts. At minimum, results should record the source dataset, original artifact path, database and table, and any relevant record identifier. Where timestamps are transformed, retain both the original and normalized values. This ensures that automated results can always be traced back to the underlying evidence.

The following Python example demonstrates how you can automatically locate the `sms.db` artifact across multiple backups. The script scans each backup directory, opens the `Manifest.db` database, and searches for entries corresponding to the SMS database:

```
import os
import sqlite3
BACKUP_ROOT = "backups/"   # Folder containing multiple iOS backups
for backup in os.listdir(BACKUP_ROOT):
    backup_path = os.path.join(BACKUP_ROOT, backup)
    manifest = os.path.join(backup_path, "Manifest.db")
    if not os.path.exists(manifest):
        continue
    conn = sqlite3.connect(f"file:{manifest}?mode=ro", uri=True)
    cursor = conn.cursor()
    query = """
    SELECT fileID
    FROM Files
    WHERE relativePath LIKE '%sms.db%'
    """
cursor.execute(query)
    result = cursor.fetchone()
    if result:
        fileID = result[0]
        folder = fileID[:2]
        sms_path = os.path.join(backup_path, folder, fileID)
        if os.path.exists(sms_path):
```

```
            print(f"Found sms.db in backup {backup}: {sms_path}")
    conn.close()
```

When executed, the script prints the path to the `sms.db` file within each backup where the artifact is present. Because iOS backups store files inside subdirectories named after the first two characters of the SHA1 identifier, the script reconstructs the correct file path automatically.

As shown in *Figure 6.5*, the script reports each backup in which `sms.db` is found and displays the reconstructed path to the corresponding database file:

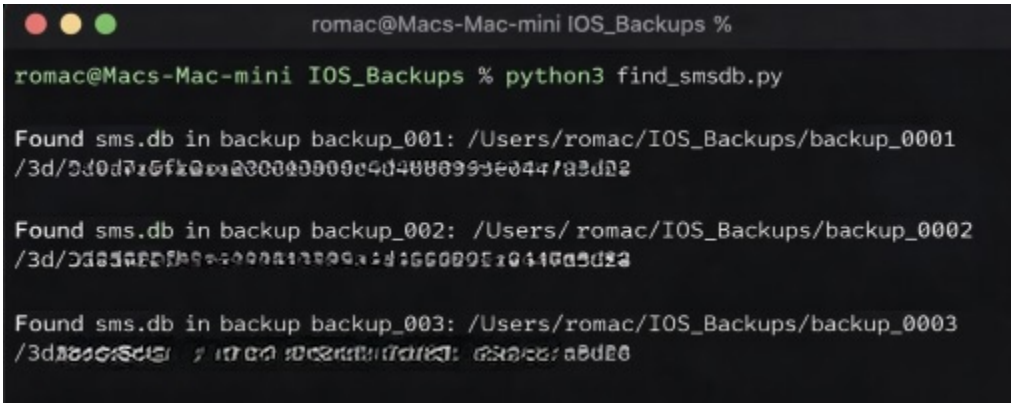


Figure 6.5: Automated identification of the SMS database across multiple iOS backups

This type of automation becomes particularly useful when analyzing historical backups. For example, you may want to determine whether a conversation appears in an earlier backup but is absent from a later snapshot. By extracting the SMS database from each backup automatically, you can compare message records across different snapshots and identify when particular artifacts appeared or disappeared.

Automated timeline construction and correlation

One of the most valuable applications of automation in iOS forensics is the construction of unified timelines from multiple artifacts. Individual data sources such as messages, call logs, calendar events, browser history, and application activity, each provide only a partial view of user behavior. When examined in isolation, their evidentiary value may be limited. When these artifacts are normalized and merged chronologically, they can reveal patterns, sequences, and contextual relationships that may be difficult to observe when each artifact is examined independently.

Automation enables this process by extracting relevant fields from different databases, converting timestamps into a consistent format, and combining records into a single, ordered

dataset. This approach allows you to correlate actions across applications and reconstruct sequences of events with greater clarity.

The following example demonstrates a simple automated workflow that combines SMS messages and calendar events into a unified timeline. These two artifacts are particularly well suited for correlation, as they often reflect communication and planned activity occurring in close temporal proximity:

```
import sqlite3
from datetime import datetime, timezone
timeline = []
# --- SMS ---
try:
        sms_conn = sqlite3.connect("file:sms.db?mode=ro", uri=True)
    sms_cursor = sms_conn.cursor()
    sms_cursor.execute("""
    SELECT date, text
    FROM message
    WHERE text IS NOT NULL
    LIMIT 10
    """)
    for row in sms_cursor.fetchall():
        ts = row[0] / 1000000000 + 978307200
        timeline.append((datetime.fromtimestamp(ts, tz=timezone.utc), "SMS",
row[1]))
    sms_conn.close()
except Exception as e:
    print(f"SMS error: {e}")
# --- Calendar ---
try:
    cal_conn = sqlite3.connect("file:Calendar.sqlitedb?mode=ro", uri=True)
    cal_cursor = cal_conn.cursor()
    cal_cursor.execute("""
    SELECT summary, start_date
    FROM CalendarItem
    WHERE summary IS NOT NULL
    LIMIT 10
    """)
    for row in cal_cursor.fetchall():
        ts = row[1] + 978307200
        timeline.append((datetime.fromtimestamp(ts, tz=timezone.utc), "SMS",
```

```
row[1]))
    cal_conn.close()
except Exception as e:
    print(f"Calendar error: {e}")
# --- Merge and sort ---
timeline.sort(key=lambda x: x[0])
# --- Output ---
for entry in timeline:
    print(f"{entry[0]} | {entry[1]} | {entry[2]}")
```

As shown in *Figure 6.6*, the script combines SMS messages and calendar events into a single timeline, with each entry displaying the normalized timestamp, artifact type, and corresponding message or event content:

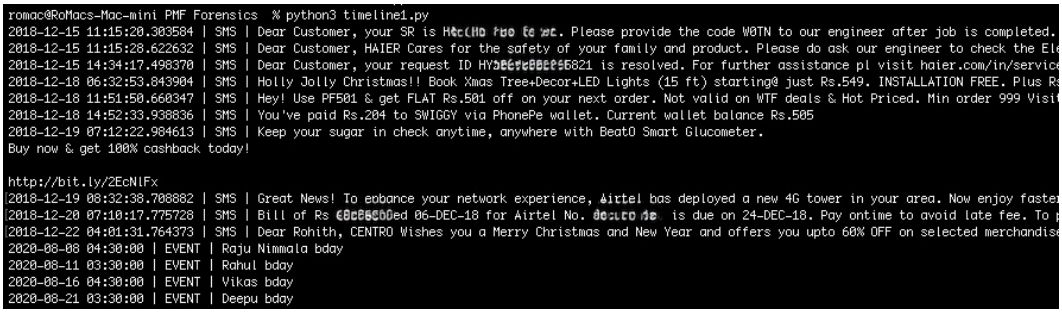


Figure 6.6: Unified chronological timeline generated from SMS messages and calendar events

By merging the normalized records chronologically, the resulting timeline allows communication activity to be examined alongside scheduled events and other time-based artifacts. In practice, such timelines can reveal coordination behavior, such as messages sent immediately before or after meetings, changes in plans, or communication spikes surrounding key events. These correlations can provide useful investigative context and help identify sequences of activity that warrant closer examination.

Note

Events appearing close together in a timeline should not automatically be assumed to be related. Temporal proximity indicates correlation, not causation. A direct relationship should only be established when supported by additional evidence.

Automation does not eliminate the need for interpretation. The selection of fields, handling of timestamps, and inclusion or exclusion of specific records all influence the final output. You must therefore understand how the timeline is constructed and be able to trace each entry

back to its source artifact when required. When implemented carefully, automated timeline construction transforms fragmented data into a coherent narrative. It allows you to move beyond isolated artifact analysis and toward a structured understanding of user behavior across time.

Automation best practices

Automation can improve the efficiency and consistency of iOS forensic analysis, but automated workflows must remain reproducible, traceable, and subject to examiner validation. The following practices provide a useful checklist:

- **Validate before scaling**: Test scripts and automated workflows against known or manually verified data before applying them across large datasets.
- **Preserve provenance:** Ensure extracted results can be traced back to their source dataset, original artifact path, database, table, and relevant record identifier.
- **Account for schema changes**: Do not assume database structures remain consistent across iOS or application versions. Scripts should identify and report unexpected schemas rather than silently producing incomplete results.
- **Preserve raw values**: Retain original values alongside transformed data, particularly when converting timestamps or decoding application-specific fields. This allows transformations to be independently verified.
- **Handle errors explicitly**: Record missing files, failed queries, parsing errors, unsupported schemas, and skipped artifacts. Automated workflows should never fail silently.
- **Protect source evidence**: Run automation against forensic working copies and use read-only access where practical to prevent unintended modification of source artifacts.
- **Document the workflow**: Preserve the scripts, queries, configuration settings, tool versions, and relevant dependencies used during analysis so that results can be reproduced.
- **Revalidate after changes**: Re-test automated workflows following significant iOS, application, forensic tool, or parser updates to ensure that previous assumptions remain valid.

With these practices in place, automation can reduce repetitive work while preserving the validation, traceability, and examiner oversight required for defensible forensic analysis.

Summary

This chapter examined how modern iOS forensic investigations rely on a combination of specialized tools and controlled automation to manage growing data complexity. As iOS ecosystems have evolved, manual analysis alone is no longer sufficient. Forensic tools play a critical role in acquiring, parsing, and presenting large volumes of structured data, but their output must always be understood as interpretation rather than direct evidence.

We explored how different categories of tools (commercial platforms and manual utilities) complement each other in practice. While automated tools accelerate artifact discovery and reporting, manual validation remains essential for verifying timestamps, database relationships, and recovered records. This balance between efficiency and accuracy defines professional forensic workflows.

The chapter also introduced practical automation techniques, including scripted artifact extraction, batch processing across datasets, and timeline construction. These approaches help standardize analysis, reduce repetitive effort, and enable you to correlate activity across multiple sources. At the same time, we emphasized that automation must be carefully validated to avoid propagating errors at scale. Finally, we outlined best practices for using tools and automation responsibly, highlighting the importance of validation, documentation, repeatability, and adaptability to evolving iOS structures. In the next chapter, we will move away from iOS to Android platforms and how they work in forensic investigations.

Part 2

Android Forensics

This section provides a comprehensive look into Android device forensics, focusing on modern Android smartphones and the evolving Android operating system. As Android continues to strengthen its security posture with features such as File-Based Encryption (FBE), Security-Enhanced Linux (SELinux) in enforcing mode, Android Verified Boot (AVB), and increasingly restrictive storage models, digital investigations involving Android devices have become more complex and constraint-driven.

You will explore a wide range of forensic acquisition techniques used to extract data from Android devices under different conditions. Given the diversity of Android hardware and software implementations, this section emphasizes understanding device states, security restrictions, and selecting appropriate extraction strategies.

This section includes the following chapters:

- *Chapter 7, Understanding Android Architecture*
- *Chapter 8, Android Forensic Setup and Pre-Data Extraction Techniques*

Whether you are working with a locked device with a secured bootloader, a partially accessible device with debugging enabled, or a rooted device in a controlled environment, this section of the book will help you understand how to acquire data from Android devices in a manner that is effective, defensible, and aligned with modern forensic practices.

7

Understanding Android Architecture

In the previous chapters, we explored iOS forensics in detail, including its architecture, filesystem, key artifacts, and acquisition and analysis techniques. In this chapter, we shift our focus to the **Android** platform and the unique challenges it presents in a forensic investigation.

Unlike iOS, the Android ecosystem is highly diverse, with variations across device manufacturers, hardware configurations, and software implementations. This chapter uses Android 17 as the primary baseline for describing Android architecture and security. Where relevant, differences in earlier Android versions are identified. Android 17 incorporates security mechanisms such as file-based encryption (FBE), application sandboxing, Security-Enhanced Linux (SELinux), and Android Verified Boot (AVB). These protections safeguard user data but also influence when, how, and to what extent forensic evidence can be accessed.

For a forensic investigator, understanding Android architecture is not just about learning how the operating system is structured. It is about understanding how security controls, storage models, and system design decisions influence data accessibility. Many common challenges in Android forensics, such as restricted access to application data, limitations of extraction techniques, and the impact of device lock state, can only be properly understood through this architectural foundation.

In this chapter, we will cover the following topics:

- Evolution of Android and its forensic impact
- Android architecture overview
- Application execution environment
- Android security model
- Encryption in Android
- Hardware-backed security mechanisms
- Verified Boot and system integrity
- Android storage and filesystem layout
- Filesystems in Android
- From Android architecture to acquisition strategy

This chapter provides a forensic-oriented view of Android, focusing on the components that matter most during an investigation. It explains how Android is structured, how applications are executed and isolated, how data is stored and protected, and how these factors influence forensic acquisition and analysis.

Evolution of Android and its forensic impact

Android is a Linux-based mobile operating system developed primarily by Google, in collaboration with the Open Handset Alliance (OHA), a consortium of hardware and software companies. Since its public release in 2008, Android has grown into one of the most widely used operating systems in the world, powering billions of devices across a wide range of manufacturers, price segments, and use cases.

In its early years, Android was designed with a strong emphasis on usability, flexibility, and rapid adoption. Device manufacturers were given significant freedom to customize the operating system, and application developers could interact with system resources with relatively fewer restrictions. From a forensic perspective, this environment was comparatively favorable. Data stored on the device was often accessible, encryption was not widely enforced, and many acquisition techniques allowed investigators to obtain a large portion of the filesystem with minimal resistance.

However, this openness also introduced security risks. As Android adoption increased, so did the number of malicious applications, data breaches, and privacy concerns. In response, the Android platform began evolving to incorporate stronger security controls aimed at protecting user data and ensuring system integrity. This marked a gradual but significant shift in how the operating system was designed and how data could be accessed.

One of the earliest changes involved strengthening **application isolation**. Applications were no longer allowed to freely interact with each other's data. Instead, each application was placed in a controlled environment, limiting its ability to access information belonging to other applications or the system. Alongside this, the permission model was refined, requiring users to explicitly grant access to sensitive data and device features. These changes improved security, but they also meant that the availability of forensic artifacts could vary depending on how applications were configured and what permissions were granted.

As the platform continued to evolve, data protection became a central focus. **Encryption mechanisms** were introduced and later expanded to ensure that data stored on the device remained protected even if the physical device was accessed. This was a major turning point for forensic investigations. Accessing raw storage no longer guaranteed access to usable data, as much of it would be stored in an encrypted form.

In more recent versions of Android, the focus has shifted even further toward **privacy and controlled data access**. Applications are now restricted in how they access shared storage, and direct access to application-specific directories is limited. System components that were once accessible for analysis, such as logs and background activity data, are now protected or restricted. These changes are designed to prevent misuse of data, but they also reduce the visibility available to forensic investigators.

Another important aspect of Android's evolution is the **diversity** of its ecosystem. Unlike platforms that are tightly controlled by a single vendor, Android runs on devices from numerous manufacturers, each introducing their own customizations. Differences in hardware, firmware, and security implementations can lead to variations in how data is stored and how accessible it is. As a result, forensic techniques that work on one device may not work on another, even if both devices are running similar versions of Android.

With respect to forensics, this evolution represents a clear transition. Earlier Android devices allowed broad and often unrestricted access to data, making acquisition relatively straightforward. In contrast, modern Android devices enforce strict boundaries around data access, requiring investigators to carefully evaluate the device state, security settings, and available access methods before proceeding.

This shift has also influenced the tools and techniques used in Android forensics. Traditional approaches such as full physical extraction are no longer always feasible. Instead, investigators increasingly rely on a combination of logical acquisition, targeted data extraction, and artifact-level analysis. Understanding the evolution of Android helps explain why these changes have occurred and prepares investigators to choose the most appropriate approach for a given scenario.

In the next section, we will examine Android architecture in detail and understand how its layered design supports both system functionality and security, and how it directly influences forensic acquisition and analysis.

Android architecture overview

To effectively perform forensic analysis on Android devices, it is essential to understand how the operating system is structured. Android follows a layered architecture, where each layer is responsible for specific functions and provides services to the layers above it. This design allows the operating system to remain flexible, scalable, and adaptable across a wide range of devices and hardware configurations.

From a forensic angle, these layers are not just conceptual components. Each layer plays a role in how data is created, stored, accessed, and protected. Understanding these layers helps you to determine where evidence resides, how it can be accessed, and what restrictions may apply during acquisition and analysis. At a high level, Android can be represented as a layered software stack consisting of the Linux kernel, hardware abstraction layer, native libraries and Android Runtime, application framework, and applications.

Figure 7.1 presents a simplified conceptual view of these layers; implementation details vary across Android versions and device manufacturers:

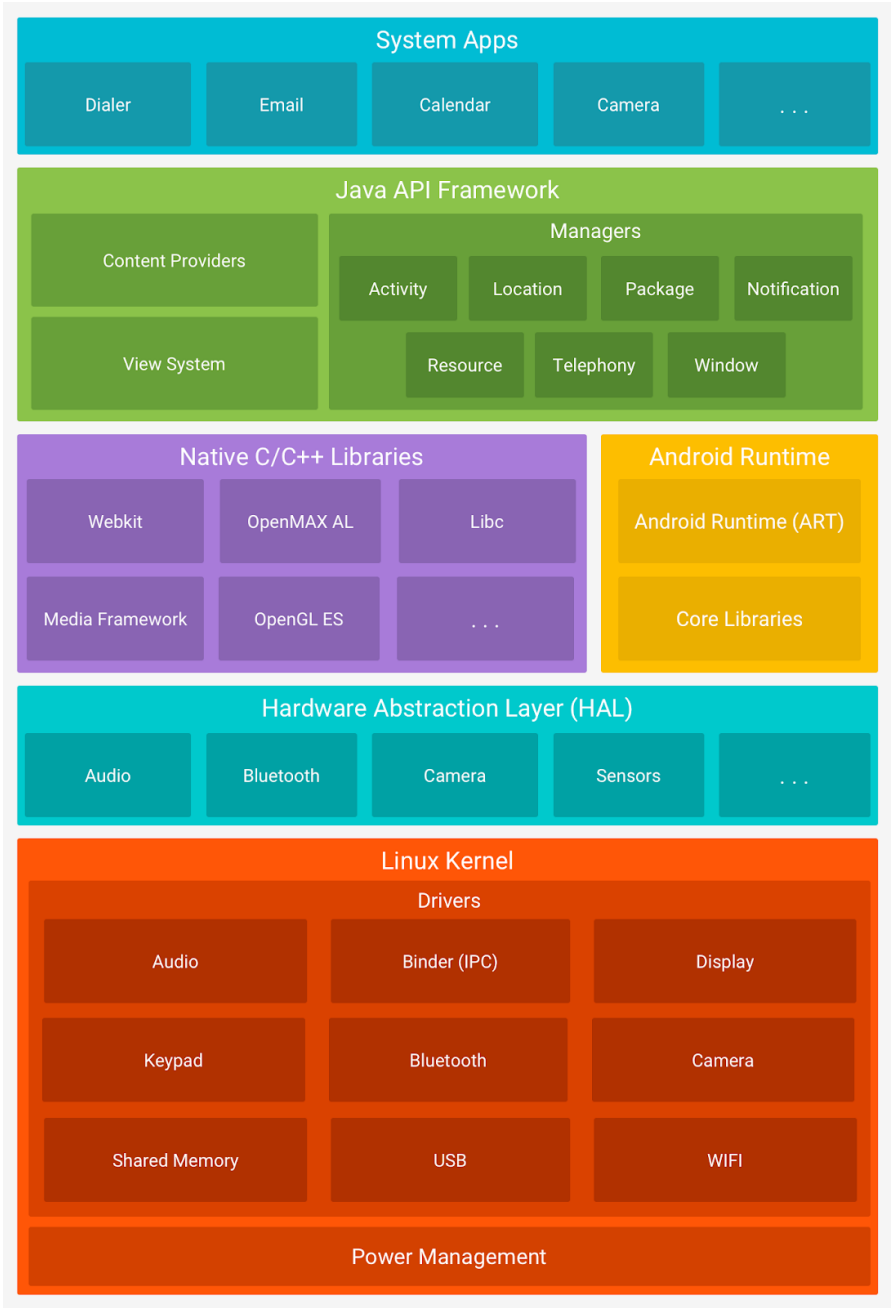


Figure 7.1: Android architectural layers

Linux kernel layer

At the foundation of the Android operating system lies the **Linux kernel**. The kernel acts as a bridge between the hardware and the software components of the system. It is responsible for core functions such as process management, memory management, device drivers, networking, and security enforcement.

The kernel plays a crucial role in Android security. It enforces process isolation, ensuring that each application runs independently. It also manages permissions at a low level, preventing unauthorized access to system resources and data. Features such as user-based access control and process separation originate from the Linux kernel and form the basis of Android's security model.

From a forensic standpoint, the kernel defines the lowest level at which data access can be controlled or restricted. Even if higher-level protections are bypassed, kernel-level enforcement mechanisms can still prevent access to certain parts of the system. Additionally, access to kernel-level data structures and memory is typically limited, making live memory acquisition more challenging on modern devices.

Hardware abstraction layer (HAL)

Above the kernel lies the **hardware abstraction layer** (**HAL**). The HAL provides a standardized interface between the hardware components and the higher-level system services. It allows Android to communicate with device-specific hardware such as cameras, sensors, and radios without requiring changes to the upper layers of the operating system.

For forensic investigators, the HAL is generally not a direct source of evidence. However, it plays an indirect role in how data is generated and processed. For example, data from sensors, cameras, and communication modules flow through this layer before being used by applications. Understanding this interaction can be useful when interpreting how certain artifacts, such as images or location data, are created.

Native libraries and Android Runtime (ART)

The next layer consists of **native libraries** and **Android Runtime** (**ART**). Native libraries are written in C or C++ and provide essential functionalities such as database management, media processing, graphics rendering, and secure communications. One of the most important libraries is SQLite, which is widely used by applications to store structured data.

ART is responsible for executing applications. Applications are compiled into a format that can be efficiently executed on the device, improving performance and reducing resource consumption. ART provides the managed runtime environment for Android applications, including bytecode execution, memory management, and garbage collection.

This layer is highly significant because many application artifacts are stored in formats and structures managed by these libraries. For example, application data stored in SQLite databases, cached files, and compiled code artifacts all originate from this layer. Understanding how applications are executed and how data is handled at runtime helps you interpret the artifacts extracted from the device.

Application framework

The **application framework** provides the higher-level services that applications rely on to function. It includes components responsible for managing user interfaces, handling system resources, managing notifications, and enabling communication between applications.

Key components of the framework include content providers, activity managers, and package managers. Among these, **content providers** are particularly important in a forensic context. They act as structured interfaces through which applications can share data with other applications. Many forensic tools leverage these interfaces to extract data without requiring direct access to application storage.

The application framework also enforces many of the rules related to permissions and data access. It determines how and when applications can access sensitive information such as contacts, messages, and location data.

System applications and user applications

At the top of the architecture are the applications themselves. These include both **system applications** that come preinstalled on the device and **user-installed applications** downloaded from app stores or other sources. This layer is the primary source of evidence. User activities, communications, media files, and application usage data are all generated and stored by these applications. Examples include messaging apps, browsers, social media platforms, and financial applications.

Each application operates within its own isolated environment, with its own storage space and permissions. This isolation enhances security but also makes forensic acquisition more complex, as accessing data across multiple applications often requires elevated privileges or specialized techniques. Understanding these layers as a cohesive system is essential for effective Android forensics as each layer contributes to how data is handled and protected, and together they define the boundaries within which forensic investigations must operate.

In the next section, we will examine how applications are executed within this architecture and how their execution model influences data storage and accessibility.

Application execution environment

Understanding how applications are executed on Android devices is essential for forensic analysis. Most of the data that you rely on originates from applications, including user communications, browsing activity, cached content, and configuration files. To interpret this data correctly, it is important to understand how applications are structured, executed, and isolated within the Android environment.

Dalvik vs ART (evolution and current state)

In earlier versions of Android, applications were executed using the **Dalvik Virtual Machine** (**DVM**). Application code was compiled into Dalvik Executable (DEX) bytecode, a format designed specifically for Android. Dalvik primarily relied on just-in-time (JIT) compilation, where frequently executed portions of the application could be compiled into native code while the application was running.

Starting with Android 5.0, the **Android Runtime** (**ART**) replaced Dalvik as the default runtime. Early versions of ART primarily used ahead-of-time (AOT) compilation, compiling application bytecode during installation. Starting with Android 7.0, ART introduced a hybrid approach that combines AOT compilation, JIT compilation, and interpretation. Runtime profiling identifies frequently executed code, which can subsequently be optimized using profile-guided compilation. This hybrid approach continues in modern Android versions, balancing application performance, compilation time, and storage usage.

Figure 7.2 illustrates the evolution from Dalvik's primarily JIT-based execution model to the AOT-focused implementation of early ART and the hybrid execution model used by modern ART:

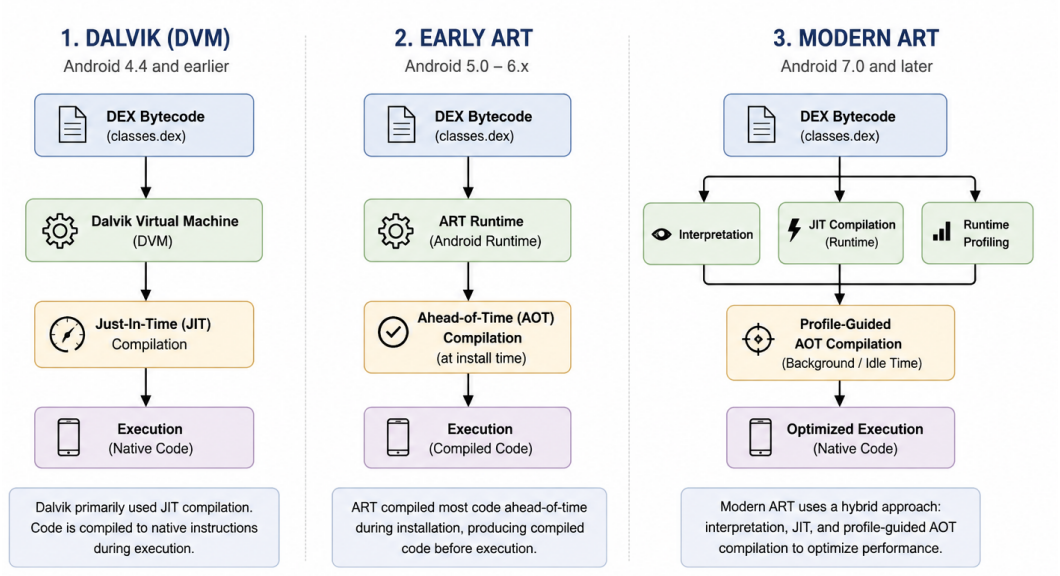


Figure 7.2: Evolution of Android application execution from Dalvik to modern ART

The transition from Dalvik to ART also affects the application artifacts you may encounter. Android applications are commonly distributed as APK files containing application code, resources, configuration information, and other components. Application bytecode is stored in DEX files, which remain important when examining or reverse engineering Android applications. ART continues to execute DEX bytecode even though the way that code is optimized and executed has evolved.

When an application is installed and executed, ART may generate additional artifacts as part of verification, optimization, profiling, and compilation. Depending on the Android version and device configuration, you may encounter files such as `.odex`, `.vdex`, `.art`, and runtime profile files. For example, `.odex` files may contain AOT-compiled application code, while `.vdex` files contain information used during verification and optimization. Runtime profile files can contain information used to guide subsequent compilation and optimization.

These artifacts can be useful when examining how an application was installed, optimized, or executed, as well as during application reverse engineering. However, their exact locations, formats, and contents can vary between Android versions. You should therefore interpret these artifacts in the context of the Android version being examined rather than assuming that the same ART artifacts will be present on every device.

Tip

When reverse engineering an Android application, start with the DEX files from the APK rather than assuming that optimized runtime artifacts are the application's original executable code. ART may create additional compiled artifacts as part of installation and execution.

Beyond the executable and optimization artifacts generated by ART, applications also create and maintain data within dedicated storage environments. The security mechanisms that isolate these environments are examined later in the *Android security model* section; the following subsection focuses on the types of data applications store.

Application storage model

Android's application-isolation model extends to data storage. Each application is assigned a private storage area associated with its unique user identifier (UID), allowing it to maintain databases, preferences, cached content, configuration files, and other application-generated artifacts without exposing them directly to unrelated applications. This private area often contains the most valuable application evidence, but access is normally restricted by Linux permissions, application sandboxing, and SELinux policies.

Applications may also create files in shared storage, particularly media, documents, downloads, and exported content intended to be accessible outside the application. These files may supplement evidence recovered from private application storage, although they often provide less context when examined independently. The availability of both private and shared data depends on the acquisition method, privileges obtained, Android version, and storage permissions in effect.

You should also recognize that application evidence can span several formats and related files. SQLite databases, XML preference files, binary data, caches, attachments, and database companion files may collectively be needed to interpret an application's activity. Finding one database or directory should therefore not be treated as a complete representation of the application's stored evidence.

The specific storage paths, partition structure, scoped-storage restrictions, and filesystems used by Android are examined in the chapter. Together, Android's application execution, isolation, and storage mechanisms determine how application evidence is created and accessed during an investigation. Access to that evidence is also governed by Android's broader security architecture, which is discussed next.

Android security model

Android has evolved into a platform with a strong focus on security. Over time, it has incorporated multiple layers of protection that control how applications interact with the operating system, hardware, and each other. These controls are intended to protect user data and system integrity, but they also directly impact how forensic investigations are performed.

The **Android security model** defines the boundaries of data accessibility and determines what data can be accessed, under what conditions, and using which techniques. Understanding these mechanisms is essential for selecting appropriate acquisition methods and correctly interpreting the data obtained from a device.

Linux permission model and UID/GID

At its core, Android inherits its security foundation from the Linux operating system. Each application is assigned a UID and operates as a separate user within the system. This model ensures that applications are isolated from one another at the operating system level. File access is controlled using standard Linux permissions based on user and group ownership. An application can only access files that belong to its assigned UID unless explicit permissions are granted. This prevents applications from directly accessing the data of other applications.

In forensics, this model is significant because it limits direct access to application data. Without elevated privileges, an investigator cannot simply browse all application directories and retrieve their contents. This is one of the primary reasons why advanced acquisition techniques are often required.

Application sandbox enforcement

One of the fundamental principles of Android security is application isolation. Android normally assigns each application a distinct Linux UID and runs it within a separate process and sandbox. Application code executes using Android Runtime (ART), while Linux permissions and SELinux policies restrict access to other applications' processes and private files.

For the primary device user, private application data is commonly stored under paths such as `/data/user/0/<package_name>/`. Applications cannot normally access another application's private data directly, although controlled communication may occur through permission-protected mechanisms such as Binder IPC, services, and content providers.

Figure 7.3 illustrates these boundaries:

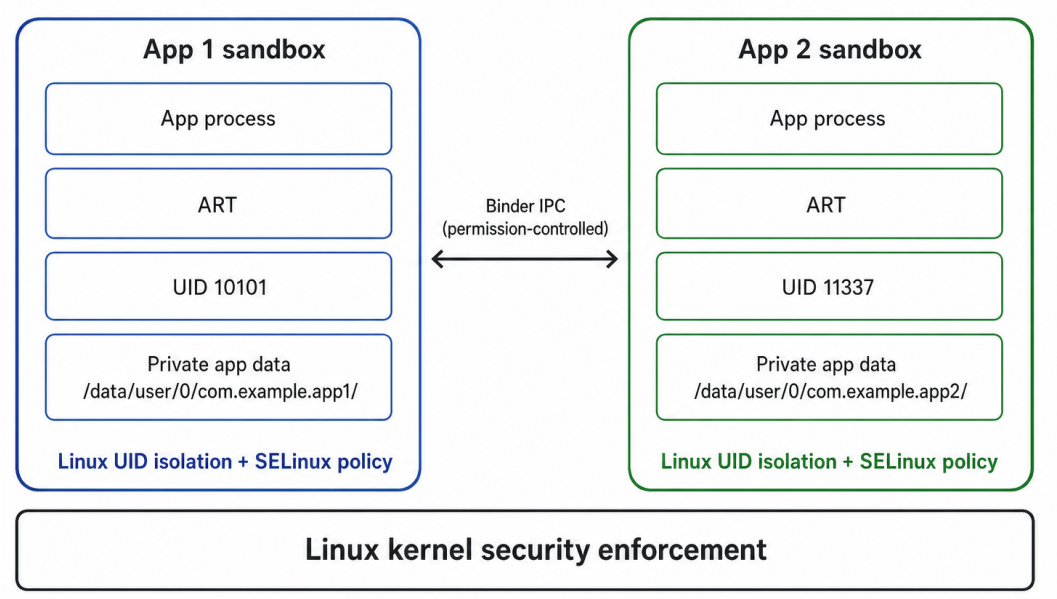


Figure 7.3: Application sandboxing and permission-controlled communication in modern Android

This isolation has important forensic implications. Without elevated privileges or a supported acquisition mechanism, you cannot simply browse every application's private storage. Evidence is distributed across separately protected application sandboxes and user or profile directories, and the visibility of each environment depends on factors such as the acquisition method, Android version, device state, privileges obtained, and applicable SELinux policies. Access to one application's data therefore does not imply access to data belonging to other applications.

Permission model (install-time vs. runtime permissions)

Android uses a permission-based model to control access to sensitive device features and user data. In earlier versions of Android, permissions were granted at the time of application installation. Users had to accept all requested permissions to install the application. Starting from Android 6.0, the **permission model** shifted to a **runtime approach**. Applications request permissions during execution, and users can grant or deny them individually. Users can also revoke permissions at any time through system settings.

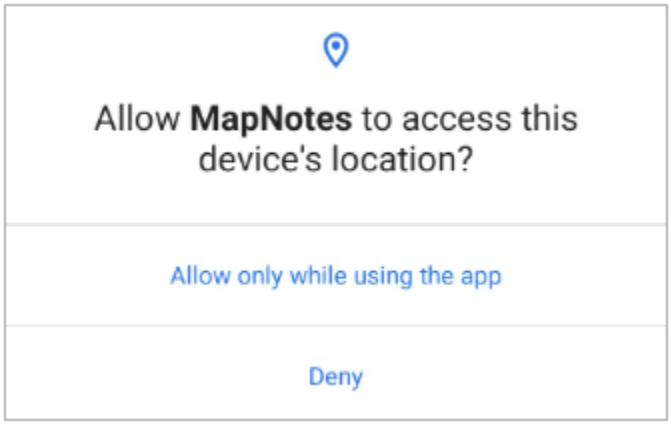


Figure 7.4: Android Runtime permission dialog

From a forensic perspective, permission-state can help explain why expected data is present or absent. For example, an application without permission to access contacts may not have imported contact names or other contact-derived information. You should therefore consider the application's permissions, Android version, and data sources when interpreting its stored artifacts.

Securing inter-process communication (IPC)

Android provides mechanisms for applications to communicate with each other in a controlled manner. These mechanisms include components such as intents, services, and content providers. They allow applications to share data and functionality without violating the isolation enforced by the sandbox. Among these, content providers are particularly relevant for forensic analysis.

Content providers give structured access to application data and are often used by forensic tools to extract information without requiring direct filesystem access. However, access to these communication channels is also controlled by permissions and system policies. Not all data exposed through these mechanisms is accessible, and restrictions may vary depending on the device and Android version.

Application signing and trust model

All Android applications must be digitally signed before they can be installed on a device. Android uses the application's **signing certificate** to verify application updates and establish continuity between versions of the same application. An update is normally accepted only when its signing identity is compatible with the version already installed on the device.

Note

Historically, Android also allowed applications signed with compatible certificates to use the same Linux UID through the `android:sharedUserId` manifest attribute. Applications sharing a UID could access each other's data and, in some cases, run within the same process. However, `sharedUserId` was deprecated in Android 10 (API level 29) and is strongly discouraged for modern applications. Existing applications may still use it for compatibility, but newly developed applications should instead communicate through controlled mechanisms such as services, content providers, and other explicitly exposed application components.

Application signing remains important during forensic analysis because signing information can help establish relationships between application packages, verify whether an APK is consistent with an expected publisher or installed version, and identify potentially modified or repackaged applications. When examining older applications, you should also be aware of legacy shared UIDs because they can affect application isolation and data accessibility.

Security-Enhanced Linux (SELinux)

Security-Enhanced Linux (**SELinux**) is a critical component of the Android security model. It enforces mandatory access control policies that define how processes can interact with each other and with system resources. Unlike traditional permission models, where access is granted unless explicitly denied, SELinux follows a strict policy where actions are denied unless explicitly allowed.

This significantly reduces the attack surface and prevents unauthorized operations, even if an application has elevated privileges. SELinux operates in enforcing mode on modern Android devices, meaning that policy violations are actively blocked rather than just logged. This adds an additional layer of protection beyond standard Linux permissions and application sandboxing. For forensic investigators, SELinux presents a significant challenge. Even with elevated access, certain operations may still be restricted by SELinux policies. This limits the effectiveness of some traditional techniques and requires them to adapt their approach based on the device's security configuration.

The Android security model is a layered system that combines multiple mechanisms to protect data and enforce system integrity. Each of these mechanisms contributes to the overall security of the platform but also introduces constraints that must be considered during forensic investigations. In the next section, we will examine how Android protects data through encryption and how this impacts forensic acquisition and analysis.

Encryption in Android

Encryption plays a central role in protecting user data on modern Android devices. It ensures that even if the physical storage of a device is accessed, the data remains unreadable without proper authentication. Over time, Android's approach to encryption has evolved significantly, moving from device-wide protection to more flexible and granular models. Encryption is one of the most important factors influencing data accessibility. The presence of encryption does not just protect data; it determines when and under what conditions that data can be accessed during an investigation.

Full disk encryption (FDE)

Full-disk encryption (**FDE**) was the principal storage-encryption model on many earlier Android devices. Android introduced FDE in Android 4.4 and expanded its use beginning with Android 5.0. Under FDE, the user data partition is protected as a single encrypted volume. The encryption key is protected using the user's device credential and, on supported devices, hardware-backed mechanisms. Before the necessary credential and key material become available after boot, encrypted user data remains inaccessible.

Android began transitioning away from FDE when file-based encryption (FBE) was introduced in Android 7.0. Devices launched with Android 10 or later are required to use FBE. Android 10 through Android 12 retained FDE support only for devices originally launched with Android 9 or earlier and subsequently upgraded. Android 13 removed FDE support entirely. Investigators are therefore most likely to encounter FDE on older or legacy devices.

File-based encryption (FBE)

Android introduced **FBE** in Android 7.0. Unlike FDE, which protects the user data partition as a single encrypted volume, FBE allows different files to be protected using different encryption keys that can be unlocked independently. Devices launched with Android 10 or later are required to use FBE, making it the relevant storage-encryption model for modern Android devices.

FBE also enabled Android's **Direct Boot** model. Data can be placed into two primary storage categories: **device encrypted** (**DE**) **storage** and **credential encrypted** (**CE**) **storage**. DE storage becomes available after the device boots and can therefore support limited functionality before the user unlocks the device. CE storage, which normally contains more sensitive user data, becomes available only after the user unlocks the device and the required credentials become available.

This distinction is particularly important during forensic acquisition. A device that is powered on but has not been unlocked after boot may expose a different set of data than a device that

has already been unlocked. You must therefore consider the device's boot and lock state when selecting an acquisition method and interpreting the data that can be accessed.

Modern Android devices may also use **metadata encryption** in conjunction with FBE. Metadata encryption protects filesystem information such as filenames, file sizes, ownership, permissions, and extended attributes. Consequently, obtaining raw encrypted blocks may not reveal meaningful file contents or complete filesystem context without the required keys.

Device encrypted (DE) storage

Under FBE, some data is stored in device encrypted storage, which becomes available to the operating system after boot without requiring the user to unlock the device. The data is protected using keys that are available to the system without requiring user authentication. This type of storage is typically used for system-level data and information required for basic device functionality. For example, certain system services and alarms may rely on this data to operate before the user unlocks the device.

From a forensic perspective, DE data may be available before the first user unlock, but its availability to the operating system does not automatically make it accessible to an examiner. Actual access still depends on the acquisition method, privileges, device implementation, and other security controls. While this data is generally less sensitive than user data, it can still provide useful information, such as system activity or device configuration details.

Credential encrypted (CE) storage

Most user data is stored in a protected category that requires user authentication to access. This data remains encrypted until the user unlocks the device with their passcode, PIN, or pattern. Once the device is unlocked, this data becomes accessible to the system and applications. However, if the device is rebooted, the data returns to its protected state until the user unlocks it again.

This distinction is critical. If a device has not been unlocked after boot, a large portion of user data will remain inaccessible, even if the investigator has access to the device's storage. This has a direct impact on acquisition strategies and may limit the amount of data that can be extracted.

Tip

Record the device state at the time of acquisition alongside the extraction itself. Two extractions from the same device can contain different artifacts simply because they were performed in different lock states.

Forensic implications of Android encryption

The introduction and evolution of encryption in Android have fundamentally changed forensic practices. Investigators can no longer assume that accessing the device storage will yield usable data. Instead, you must consider factors such as the device's lock state, whether it has been recently unlocked, and the type of encryption in use.

Encryption also reduces the effectiveness of certain traditional techniques. Methods that rely on direct access to storage may only retrieve encrypted data, which cannot be interpreted without the appropriate keys. As a result, you often need to rely on alternative approaches, such as live acquisition, logical extraction, or analysis of previously unlocked devices.

Understanding how encryption is implemented in Android is essential for determining what data may be accessible, what remains protected, and how the device state affects acquisition. However, modern Android security does not rely on encryption alone. Encryption keys and other sensitive operations are increasingly protected by dedicated hardware-backed security mechanisms, which we examine next.

Hardware-backed security mechanisms

In addition to software-based protection, modern Android devices incorporate **hardware-backed security mechanisms** to protect sensitive data. These mechanisms are designed to ensure that critical operations, such as cryptographic key management, occur in isolated environments that are resistant to compromise, even if the main operating system is attacked.

Hardware-backed security represents one of the strongest barriers to data access. Unlike software protections, which may sometimes be bypassed or analyzed, hardware-enforced protections are significantly more difficult to overcome and often make direct access to certain types of data impractical.

Android Keystore

The **Android Keystore** system provides applications and system components with controlled storage and use of cryptographic keys. Applications can generate keys for encryption, decryption, signing, and authentication operations. Keys managed through Android Keystore are generally non-exportable through the Keystore API.

Depending on the device and key configuration, cryptographic operations may be enforced in software, a **trusted execution environment** (**TEE**), or **StrongBox**. Hardware-backed keys are substantially more resistant to extraction because the cryptographic operation can occur within secure hardware without exposing the raw key material to the Android operating system.

Many applications use Keystore-managed keys to protect credentials, tokens, and locally encrypted databases. Obtaining an application's files may therefore be insufficient to decrypt its contents. However, you should avoid assuming that every Keystore key is hardware-backed or that every implementation is immune to device-specific vulnerabilities.

Tip

Finding an encrypted application database does not necessarily mean that its decryption key will be stored alongside it. Applications may protect keys through the Android Keystore, so examine the application's key-management implementation before assuming that a key can be recovered from its data directory.

Trusted execution environment (TEE)

The TEE is an isolated execution environment supported by the device's processor. It can perform sensitive operations, including cryptographic processing and hardware-backed key management, separately from the main Android operating environment. Android communicates with the TEE through controlled interfaces, allowing cryptographic operations to be performed without normally exposing raw key material to Android.

TEE-backed protection can substantially limit access to cryptographic keys even when the Android operating system is compromised. Its effectiveness nevertheless depends on the device's implementation, security patch level, configuration, and the absence of exploitable vulnerabilities in the TEE and its interfaces.

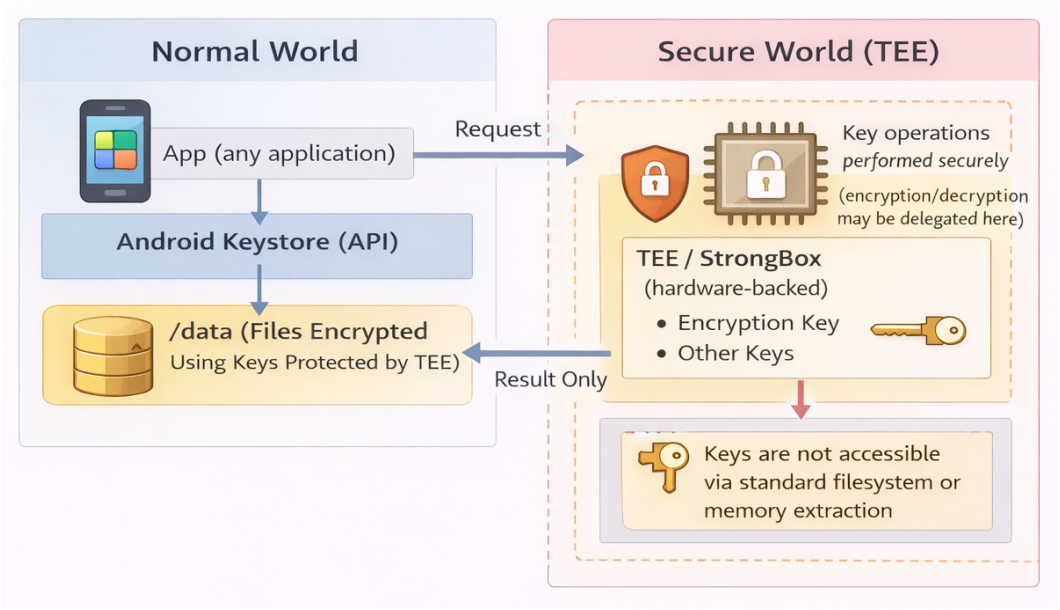


Figure 7.5: Hardware-backed Android Keystore operations in a TEE or StrongBox

TEE significantly limits the ability to access protected data. Even if the main operating system is compromised, cryptographic keys, and operations protected within the TEE may remain isolated from the Android operating environment. This can make extraction of certain information, particularly cryptographic keys used to protect application data, significantly more difficult.

StrongBox secure element

StrongBox is an optional hardware-backed security feature available on some Android devices. It is implemented in a discrete secure element that provides key storage and cryptographic operations separately from the main application processor. Compared with a typical TEE implementation, StrongBox is designed to provide stronger resistance to physical tampering and side-channel attacks. StrongBox-backed keys can therefore present a significant barrier to forensic decryption. However, you should confirm that the relevant key is Strongbox-backed rather than assuming that the presence of StrongBox means all application keys use it.

In the next section, we will examine how Android verifies system integrity during the boot process and how this affects forensic acquisition techniques.

Verified Boot and system integrity

Modern Android devices implement strong mechanisms to ensure that the operating system has not been tampered with. These mechanisms are designed to verify the integrity of the system at every stage of the boot process, prevent unauthorized modifications, and protect user data from compromise.

These protections introduce a significant constraint. Many traditional techniques that involve modifying the device, such as booting a custom image or gaining persistent elevated access, may trigger security mechanisms or result in data loss. Understanding how system integrity is enforced is therefore essential before attempting any acquisition method.

Chain of trust

Android uses a concept known as the **chain of trust** to verify the integrity of the system during boot. This process begins with a hardware root of trust embedded within the device. Each stage of the boot process verifies the integrity and authenticity of the next stage before passing control to it.

If any component in this chain fails verification, the device may refuse to boot, display a warning, or switch to a restricted mode. This ensures that only trusted and verified software is allowed to run on the device. This also means that unauthorized modifications to system components can be detected early in the boot process. Attempts to replace or alter system files, boot images, or recovery environments may not succeed without triggering these protections.

Android Verified Boot (AVB)

Android Verified Boot (**AVB**) is the implementation of the chain of trust in modern Android devices. It verifies the integrity of key system partitions, such as the boot and system partitions, before allowing the device to start. AVB ensures that the operating system has not been modified or corrupted. If verification fails, the device may prevent booting or display a warning indicating that the system integrity has been compromised. In some cases, the device may still boot but operate in a degraded or untrusted state.

AVB limits the ability to modify the device for data extraction. Techniques such as flashing custom recovery images or modifying system partitions are often restricted or blocked. This reduces the feasibility of approaches that rely on altering the device environment to gain access to data.

Bootloader locking and forensic impact

The **bootloader** is responsible for initializing the device and loading the operating system. On most modern Android devices, the bootloader is locked by default to prevent unauthorized modifications. Unlocking the bootloader allows the device to run custom software, but this process typically results in a factory reset, erasing all user data.

This is an important consideration in forensic investigations, as attempting to unlock the bootloader can lead to permanent loss of evidence. Even when unlocking is possible, it may leave traces on the device, such as flags indicating that the device has been modified. These changes can affect the integrity of the evidence and may need to be documented as part of the investigation. The state of the bootloader is therefore one of the first factors that must be assessed during an investigation.

A locked bootloader limits the available acquisition techniques, while an unlocked bootloader may provide greater flexibility but can also raise questions about prior modification and evidence integrity. Together, these security mechanisms determine how you can gain access to an Android device. Once access is established, the next step is understanding where potential evidence resides, which requires examining Android's storage and filesystem layout.

Android storage and filesystem layout

Understanding how Android organizes data on a device is essential for effective forensic analysis. While earlier sections focused on system architecture and security mechanisms, this section shifts attention to where data is actually stored and how it is structured within the device.

Knowing the storage layout allows you to identify relevant locations for artifacts, understand access limitations, and interpret extracted data correctly. Modern Android storage is designed with both usability and security in mind, which means that data is often distributed across multiple locations with varying levels of accessibility.

General partition structure

Android devices are divided into multiple partitions, each serving a specific purpose. However, not all partitions are equally important from a forensic standpoint.

The **data partition**, typically mounted at `/data`, is the critical location for investigations. It contains most user-generated and application-specific data, including databases, account information, logs, cached files, and application artifacts. In the majority of cases, this partition is the primary target during acquisition and analysis.

In contrast, **system-related partitions** such as `/system`, `/vendor`, and `/product` store operating system components, hardware-specific configurations, and device customizations.

These partitions are usually mounted as read-only and protected by system integrity mechanisms. While they generally do not contain user-generated evidence, they may still provide useful context, such as device configuration details or indications of system modification.

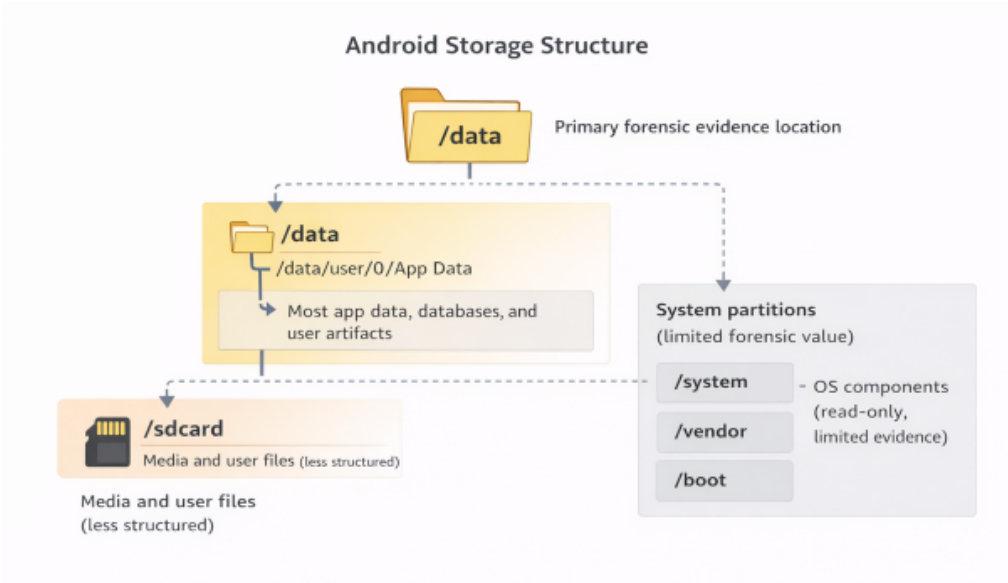


Figure 7.6: Simplified Android storage and partition structure

From a practical standpoint, forensic efforts should prioritize access to the data partition, using system partitions primarily for validation and environmental understanding.

Dynamic partitions and A/B updates

Modern Android devices commonly use **dynamic partitions**, in which logical partitions such as `/system`, `/vendor`, and `/product` are allocated within a larger super partition. This allows partition sizes to change during over-the-air updates without modifying the physical partition table.

Modern devices may also use **Virtual A/B updates**. Rather than maintaining two complete physical copies of every dynamic partition, Virtual A/B can create snapshots and copy-on-write data while an update is installed. After reboot, the updated snapshot data is merged into the underlying partitions.

These mechanisms are not primary sources of user evidence, but they can affect forensic interpretation. You may encounter logical-partition metadata, boot slots, snapshots, copy-on-write data, or an incomplete update-merge state. These should not be interpreted as ordinary duplicate user-data partitions.

Application data and user artifacts

Within the data partition, **private application data** is organized by Android user or profile and application package. For the primary user, a common location is `/data/user/0/<package_name>/`, where an application maintains databases, preferences, cache files, and other artifacts. Secondary users and managed work profiles use other numeric user IDs under `/data/user/`. You should therefore identify all users and profiles rather than limiting examination to user 0.

Modern Android devices may also contain additional isolated environments, such as Private Space, which can separate applications and their data from the primary user environment. You should identify all available users, profiles, and private environments when determining where application evidence may reside.

Related user and profile information may also be found under locations such as `/data/system/users/`, while emulated media commonly appears under `/data/media/<user_id>/` and `/storage/emulated/<user_id>/`. Exact paths and accessibility vary by Android version and manufacturer implementation.

In addition to application data, the data partition may also include system-level artifacts such as logs, usage statistics, and configuration files. These artifacts can provide valuable insight into user activity, device behavior, and application interactions.

Emulated storage (/sdcard)

Android devices provide a shared storage area that is accessible to both users and applications. This is often referred to as **external storage**, even though it may be physically located within the device. This storage is typically mounted under paths such as `/sdcard/` or `/storage/emulated/0/`.

This location is valuable because it often contains user-generated media and exported data including user-accessible files such as photos, videos, downloads, and documents. However, it is generally less structured than application-specific storage and may not contain the full context of user activity.

Scoped storage restrictions

In earlier versions of Android, applications and tools with appropriate permissions could access large portions of shared storage, including files created by other applications. This made it easier for both applications and forensic tools to browse directories and collect data. Android 10 introduced **scoped storage**, which changed how applications access files in shared storage. Android 11 strengthened and broadly enforced these restrictions.

Applications are generally limited to their own files and specific categories of shared media, with access to other locations controlled through mechanisms such as the Storage Access Framework and MediaStore. Android 11 also introduced additional restrictions on access to other applications' directories under `/sdcard/Android/data/` and `/sdcard/Android/obb/`.

For example, an application can access its own permitted storage locations but generally cannot directly read another application's private or app-specific files simply because those files are present in shared storage. Similarly, forensic tools relying on standard application permissions or storage APIs may be unable to enumerate or retrieve files from restricted directories.

These restrictions have important forensic implications. Techniques that previously relied on broad access to shared storage may produce incomplete results on Android 10 and, particularly, Android 11 and later. You may therefore need filesystem-level acquisition, elevated privileges, forensic agents, or other supported acquisition techniques to obtain data that is inaccessible through standard Android interfaces.

These storage restrictions illustrate how Android's evolving security model directly affects the visibility and accessibility of forensic evidence. Understanding these limitations is important when selecting an acquisition method and interpreting data that may appear to be missing from an extraction. The next section examines the physical, emulated, and virtual filesystems underlying these storage locations and their implications for forensic recovery and interpretation.

Filesystems in Android

The **filesystem** defines how data is stored, organized, and retrieved on a device. In Android, different partitions may use different filesystems depending on the device, hardware, and operating system version. Understanding these filesystems is important for forensic analysis, as they influence how data is written, how it can be recovered, and what artifacts may be available.

Unlike traditional desktop systems that use drive letters, Android follows a Linux-based structure where all files are organized under a single root directory (`/`). Different filesystems are mounted at specific points within this hierarchy. For example, the user data partition is mounted under `/data`, while shared storage is exposed through paths such as `/sdcard`. Identifying the filesystem in use helps determine how data behaves on disk. This includes how files are written, how deletions are handled, and whether residual data may still exist after deletion. These factors directly impact the ability to recover evidence.

You can identify the filesystems and mounted partitions on an Android device using ADB. On a device where ADB access is available, you can run:

```
adb shell cat /proc/mounts
```

```
[android # adb shell cat /proc/mounts                                                ]
tmpfs /dev tmpfs rw,seclabel,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,seclabel,relatime,mode=600,ptmxmode=000 0 0
proc /proc proc rw,relatime,gid=3009,hidepid=invisible 0 0
sysfs /sys sysfs rw,seclabel,relatime 0 0
selinuxfs /sys/fs/selinux selinuxfs rw,relatime 0 0
tmpfs /mnt tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755,gid=1000 0 0
/dev/block/vdd1 /metadata ext4 rw,seclabel,nosuid,nodev,noatime 0 0
/dev/block/dm-0 / ext4 ro,seclabel,nodev,relatime 0 0
/dev/block/dm-4 /vendor ext4 ro,seclabel,relatime 0 0
/dev/block/dm-3 /product ext4 ro,seclabel,relatime 0 0
/dev/block/dm-1 /system_dlkm erofs ro,seclabel,relatime,user_xattr,acl,cache_strategy=read
around 0 0
/dev/block/dm-2 /system_ext ext4 ro,seclabel,relatime 0 0
tmpfs /apex tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755 0 0
tmpfs /linkerconfig tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755 0 0
tmpfs /mnt/installer tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/androidwritable tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755,gid=100
0 0 0
none /dev/blkio cgroup rw,nosuid,nodev,noexec,relatime,blkio 0 0
none /sys/fs/cgroup cgroup2 rw,nosuid,nodev,noexec,relatime,memory_recursiveprot 0 0
none /dev/cpuctl cgroup rw,nosuid,nodev,noexec,relatime,cpu 0 0
none /dev/cpuset cgroup rw,nosuid,nodev,noexec,relatime,cpuset,noprefix,cpuset_v2_mode 0 0
tmpfs /linkerconfig tmpfs rw,seclabel,nosuid,nodev,noexec,relatime,mode=755 0 0
tracefs /sys/kernel/tracing tracefs rw,seclabel,relatime 0 0
/product/apex/com.google.android.gmssystem.prodvic.apex /apex/com.google.android.gmssystem
@13 erofs ro,dirsync,seclabel,nodev,noatime,user_xattr,acl,cache_strategy=readaround,fsoff
```

Figure 7.7: Mounted filesystems on an Android device

This displays currently mounted filesystems, their mount points, filesystem types, and mount options. For example, you may see entries showing partitions mounted using `EXT4` or `F2FS` and locations such as `/data`, `/system`, or `/vendor`. The exact output varies depending on the Android version, device manufacturer, and storage architecture. You can also use `adb shell df -T` to view mounted filesystems together with their filesystem type, capacity, and available space. Although forensic tools often abstract these details, examining the device directly can help you understand its storage configuration and validate the information reported by forensic tools.

Tip

Do not infer a filesystem solely from a directory path. Use the device's mount information to determine which filesystem and partition actually back a path before drawing conclusions about storage behavior or recoverability.

In addition to physical storage filesystems, Android also uses virtual and temporary filesystems such as `/proc`, `/sys`, and `tmpfs`. These do not store persistent user data but can provide valuable insight into system state, configuration, and runtime activity during analysis. Rather than covering all possible filesystem types, this section focuses on those most relevant to modern Android devices and forensic investigations.

EXT4 filesystem

EXT4 is commonly used on Android devices for system and user-data partitions. It is a journaling filesystem that records filesystem metadata changes in a journal before applying them to the main filesystem structures. This improves reliability and reduces the risk of filesystem corruption following a crash or unexpected shutdown. From a forensic perspective, the EXT4 journal may sometimes provide information about recent filesystem operations. However, journal contents are limited and continuously reused, so the journal should not be treated as a dependable source of deleted files.

EXT4 is well documented and widely supported by forensic tools, making it relatively straightforward to parse when decrypted filesystem access is available. You can examine files, metadata, and directory structures efficiently. However, deleted data recovery on modern Android devices depends on more than the filesystem format. FBE, metadata encryption, TRIM or discard operations, flash translation layers, and continued device activity may make residual data unavailable or uninterpretable.

F2FS filesystem

F2FS, or **flash-friendly file system**, is designed specifically for flash-based storage, which is commonly used in modern Android devices. It uses a log-structured design that optimizes write performance and reduces wear on flash memory. Many newer devices use F2FS for the data partition, where most user data resides, which makes it highly relevant for forensic investigations.

F2FS uses a log-structured design together with segment cleaning and garbage collection. These behaviors influence where current and stale data may reside, but they do not support a general conclusion that recoverability is always better or worse than on EXT4.

Remnants may sometimes exist in stale or unused segments, but recovery is highly dependent on filesystem state, garbage collection, TRIM or discard behavior, encryption, flash translation layers, and device activity after deletion. You should therefore avoid predicting deleted-data recovery solely from the presence of F2FS.

Emulated and virtual filesystems

Android uses several virtual or emulated filesystems to provide access to different types of data. For example, shared storage paths such as `/sdcard/` are typically implemented using a virtual layer that maps to underlying internal storage. These filesystems do not store data independently. Instead, they act as an interface that enforces access controls and presents a consistent view of storage to applications.

Other virtual filesystems, such as `/proc` and `/sys`, expose system and kernel information. The `/proc` filesystem provides details about running processes, system configuration, and kernel state, while `/sys` exposes hardware and device configuration information. These filesystems are useful for understanding device state and system behavior at a given point in time. However, they do not contain persistent user-generated data and are typically used for analysis rather than artifact recovery.

Journaling, WAL, and forensic considerations

Modern Android applications commonly use SQLite databases, many of which operate in **write-ahead logging** (**WAL**) mode. In this mode, modified database pages are written to a separate `-wal` file before they are checkpointed into the main database.

A current WAL may contain newer versions of database pages that are not yet present in the main database. Depending on transaction history, checkpointing, and page reuse, it may also preserve earlier record content. A WAL should not, however, be treated as an automatic source of deleted records.

You should acquire and examine the main database together with its associated `-wal` and `-shm` files as a consistent set. Temporary files, application caches, and filesystem journals should also be evaluated when present, but their evidentiary value depends on their state and context.

Legacy filesystems

Earlier Android devices used filesystems such as YAFFS2, which were designed for older flash storage technologies. While these filesystems were important in the early development of Android, they are rarely encountered on modern devices. Knowledge of legacy filesystems may still be useful when dealing with older devices. However, most contemporary investigations will involve EXT4 or F2FS, and investigators should focus primarily on these filesystems.

Now, let's bring together the concepts covered in this chapter and examine their overall impact on forensic investigations.

From Android architecture to acquisition strategy

Android architecture should inform the acquisition strategy selected for a device. Before attempting extraction, you should translate the protections identified in this chapter into practical questions about device state, data accessibility, and the risk of altering evidence.

The first consideration is the **device's current state**. Under FBE, credential encrypted data may remain unavailable until the user unlocks the device after boot. A powered-on device that has already been unlocked may therefore expose substantially more evidence than the same device after a restart. You should document the device's power, boot, and lock state before taking any action that could change the availability of encryption keys.

The next consideration is the **level of access** required. Logical methods operating through standard Android interfaces may recover accessible system information, shared media, and data exposed through permitted services or content providers. However, application sandboxing, Linux permissions, scoped storage, and SELinux can prevent these methods from reaching private application data. When the required evidence lies within protected application storage, you may need a supported filesystem-level method, a forensic agent, privileged access, or another device-specific acquisition technique.

System-integrity protections must also be evaluated before attempting techniques that modify the boot process or operating environment. A locked bootloader and Android Verified Boot may prevent custom images from running, while unlocking the bootloader commonly triggers a factory reset. Rooting, flashing, or booting an alternative environment may also alter the device and affect evidence integrity. These methods should therefore be considered only after their consequences are understood and less invasive sources have been evaluated.

Finally, **acquisition support** must be assessed for the specific device rather than for Android as a single platform. The device model, processor, manufacturer implementation, Android version, security patch level, filesystem, encryption configuration, bootloader state, and available authorization can all affect which techniques are feasible. A method supported on one Android device may be unavailable or unsafe on another device running the same Android version.

These considerations lead to an important interpretive principle: data absent from an extraction should not automatically be treated as nonexistent. It may instead be encrypted, isolated, excluded by the acquisition method, or unsupported by the tool. As investigators, you should document acquisition limitations and, where necessary, evaluate alternative sources

such as backups, synchronized accounts, cloud data, removable media, or other validated tools.

Summary

In this chapter, we explored the Android operating system from a forensic perspective, focusing on the components that influence how data is stored, protected, and accessed. We examined the layered architecture of Android, including the kernel, runtime, and application framework, and understood how applications are executed and isolated. The Android security model was discussed in detail, highlighting mechanisms such as sandboxing, permissions, and SELinux, which enforce strict access controls across the system.

We then explored encryption in Android and its impact on data accessibility, particularly the distinction between different types of protected data and how device state affects access. Hardware-backed security mechanisms and system integrity protections were also covered, emphasizing their role in safeguarding sensitive data and preventing unauthorized modifications. Finally, we examined how data is organized within the device, including partition structure, storage locations, and filesystems. We also discussed how modern restrictions and filesystem behavior influence data recovery and analysis.

Together, these concepts establish a foundation for understanding the constraints and challenges of Android forensics. In the next chapter, we will build on this foundation and focus on preparing the forensic environment and evaluating devices prior to data extraction.

8

Android Forensic Setup and Pre-Data Extraction Techniques

In the previous chapter, we explored the Android operating system from a forensic perspective, including its architecture, security model, storage design, and the constraints that affect data access. These concepts highlighted an important reality: modern Android devices are designed to protect user data, and these protections directly influence how forensic investigations must be conducted.

In this chapter, the focus shifts from understanding the platform to operating within it. Before any data extraction is attempted, it is critical to properly prepare the forensic environment, handle the device carefully, and evaluate what level of access is realistically achievable. Decisions made at this stage can significantly impact the success of the investigation. Actions such as rebooting a device, unlocking a bootloader, or enabling certain settings may result in data loss, encryption barriers, or permanent alteration of evidence.

Unlike earlier versions of Android, where data extraction was often straightforward, modern devices require a more structured and cautious approach. You must consider factors such as device state, lock status, encryption, and system protections before choosing an acquisition method. This chapter provides a practical, step-by-step framework to guide that process, helping you determine what actions are safe, what techniques are viable, and how to prepare the device for extraction without compromising evidence.

In this chapter, we will cover the following topics:

- Introduction to pre-extraction workflow
- Forensic workstation and tooling setup
- Handling and preparing the device for examination

- Screen lock and access considerations
- Access strategies and practical approaches
- Rooting and privilege escalation
- Working with ADB shell: Essential Linux commands
- Validation and pre-extraction checklist

Introduction to pre-extraction workflows

Modern Android forensics begins well before any data is extracted from the device. In earlier versions of Android, you could often connect a device and proceed directly with acquisition. That approach is no longer reliable. Today, Android devices enforce multiple layers of security, including encryption, application sandboxing, and system integrity protections. As a result, the success of an investigation depends heavily on the actions taken before extraction even begins.

The **pre-extraction phase** is where you assess the device, preserve its current state, and determine what level of access is realistically achievable. This phase is critical because certain actions, even those that seem routine, can have irreversible consequences. For example, rebooting a device may lock access to encrypted data, unlocking a bootloader may trigger a data wipe, and enabling or modifying system settings may alter evidentiary artifacts. These risks make it essential to approach each device with a clear and structured workflow.

A key objective during this stage is to understand the **condition of the device**. Factors such as whether the device is powered on, whether it is locked or unlocked, and whether communication interfaces such as USB debugging are enabled will directly influence the acquisition strategy. You must evaluate these conditions carefully before interacting with the device further.

Another important aspect of the pre-extraction workflow is maintaining **forensic integrity**. Every interaction with the device should be controlled, documented, and justified. This includes isolating the device from networks to prevent remote access or wiping, ensuring that the device remains in a stable state, and avoiding unnecessary changes to the system. Even seemingly minor actions can introduce changes that complicate analysis later.

Rather than focusing on tools or specific techniques, this phase emphasizes decision making. The goal is not to immediately extract data, but to determine the safest and most effective path forward. In many cases, what you choose not to do is just as important as what you do.

By the end of this phase, you should have a clear understanding of the state of the device, the constraints imposed by the platform, and the most appropriate acquisition approach. This foundation ensures that the extraction process, can be carried out in a controlled and defensible manner.

Forensic workstation and tooling setup

Before interacting with any Android device, it is essential to establish a controlled and reliable **forensic workstation**. The environment in which the examination is conducted directly influences the stability of device communication, the success of data acquisition, and the integrity of the evidence. A properly prepared workstation minimizes unexpected behavior, reduces the risk of contamination, and ensures that all actions performed during the investigation are deliberate and repeatable.

A forensic workstation should be logically separated from everyday use. Background applications, automatic synchronization services, and unnecessary utilities should be minimized or disabled. This is particularly important when handling mobile devices, as unintended interactions such as auto-mounting storage or triggering background processes may alter data on the device. Maintaining a controlled environment ensures that all interactions with the device are intentional and can be documented accurately.

Installing Android platform tools

Modern Android forensics relies on the **Android platform tools**, with the **Android Debug Bridge** (**ADB**) serving as the primary interface for communication between the workstation and the device. These tools provide a consistent and widely supported method to interact with Android devices across different manufacturers and OS versions.

ADB is a command-line utility that allows you to send commands to an Android device, retrieve data, and interact with the device's underlying operating system. It operates as a client-server architecture, where a client on the workstation communicates with a background service (daemon) running on the device. This communication enables a wide range of forensic operations, including listing files, copying data, accessing logs, and executing shell commands on the device.

From a forensic perspective, ADB is particularly valuable because it provides controlled access to the device without requiring proprietary tools. However, the level of access available through ADB depends on the device state. For example, access may be limited if the device is locked, if USB debugging is disabled, or if the connection has not been authorized by the user. These constraints will be explored in later sections of this chapter.

To prepare the environment, you must download the latest version of the Android platform tools from the official Android developer website and extract the contents to a known location on the workstation. The extracted directory contains essential binaries such as `adb` and `fastboot`, which will be used throughout the investigation.

To verify that the tools are installed correctly, open a terminal or command prompt, navigate to the `platform-tools` directory, and execute the `adb` version command. A successful response displaying version information confirms that the tools are properly installed and accessible. *Figure 8.1* shows an example of the expected output on a macOS workstation:

```
romac@RoMacs-Mac-mini ~ % adb version
Android Debug Bridge version 1.0.41
Version 37.0.0-14910828
Installed as /opt/homebrew/bin/adb
Running on Darwin 24.6.0 (arm64)
romac@RoMacs-Mac-mini ~ %
```

Figure 8.1: Output on a Mac device confirming successful adb installation

Verifying ADB connectivity

Before working with evidentiary devices, it is important to confirm that ADB is functioning as expected. This ensures that the workstation can detect and communicate with Android devices.

To verify ADB functionality, connect any sample device that has the **USB debugging** option turned on and then run the following steps:

1. Open a terminal or command prompt
2. If the `platform-tools` directory is not added to the system PATH, navigate to the directory where it is installed
3. Execute the following command: `adb devices`
4. If no device is connected, the command will return an empty list
5. When a device is connected and recognized, its identifier will appear in the output. *Figure 8.2* shows an example of the output when ADB successfully detects a connected Android device:

```
romac@RoMacs-Mac-mini ~ % adb devices
List of devices attached
3c9be2860106    device

romac@RoMacs-Mac-mini ~ %
```

Figure 8.2: adb output showing a connected device

Understanding ADB device states

When an Android device is connected to a forensic workstation, ADB reports the connection status of the device. Interpreting these states correctly is essential, as they directly determine whether interaction with the device is possible and what level of access can be achieved.

Tip

Capture the initial output of `adb devices` before attempting to resolve an offline or unauthorized state. This preserves a record of the device's original connectivity and authorization state before further interaction.

The output of `adb devices` command lists all connected devices along with their current state. Common ADB states include the following:

- **device**: This indicates that the device is connected, authorized, and ready for interaction (as seen in *Figure 8.2*). In this state, the workstation has been approved on the device, and ADB commands can be executed successfully. This is the ideal state for performing forensic acquisition, as it allows access to the device shell and enables data extraction commands.
- **unauthorized**: This state indicates that the device is connected but has not authorized the workstation for ADB access. In modern Android versions, ADB connections must be explicitly approved by the user through a confirmation prompt displayed on the device screen. Until this authorization is granted, no commands can be executed. *Figure 8.3* shows the output of the `adb devices` command when a connected device has not yet authorized the workstation:

```
romac@RoMacs-Mac-mini ~ % adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
3c9be2860106    unauthorized
```

Figure 8.3: adb output showing a connected device but not yet authorized

The authorization message that you would see on the corresponding Android device to allow USB debugging:

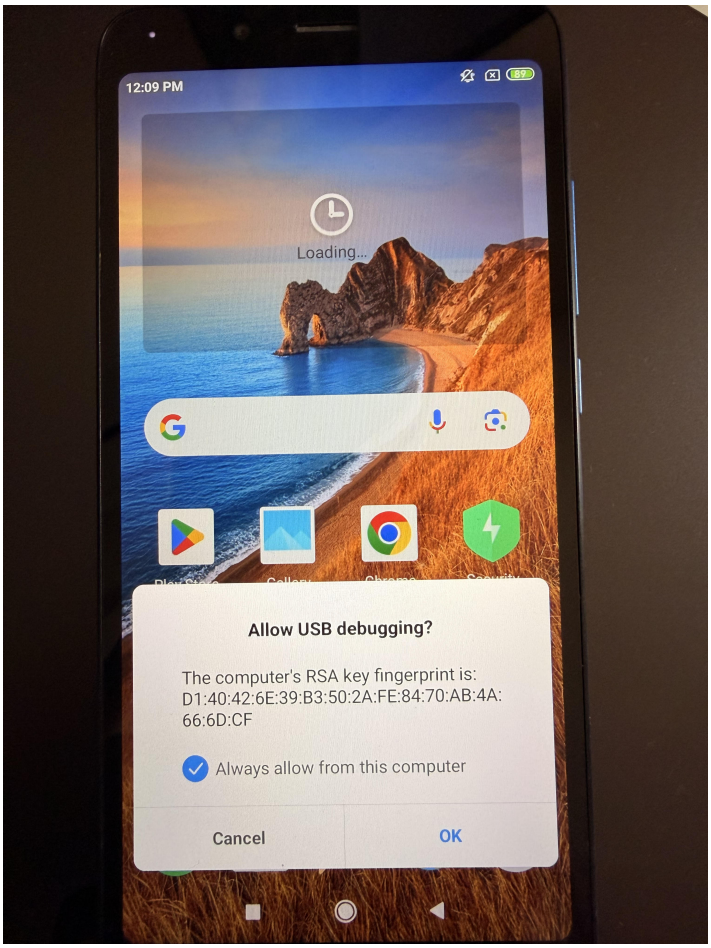


Figure 8.4: Allow USB debugging message on Android device

From a forensic perspective, this state presents a significant limitation. If the device is locked and the authorization prompt cannot be accepted, ADB access cannot be established. This highlights the importance of gaining access to an unlocked device during initial seizure whenever possible.

- **offline**: This state indicates that the device is detected but is not properly communicating with the workstation. This may occur due to driver issues, unstable USB connections, or problems with the ADB daemon. In such cases, restarting the ADB server or reconnecting the device may resolve the issue.

These connection states should always be verified before beginning acquisition. A device listed as 'device is ready for ADB interaction,' while unauthorized or offline states must be resolved before reliable acquisition can proceed.

Drivers and system considerations

The requirements for device connectivity vary depending on the operating system. On macOS and Linux systems, ADB typically functions without additional drivers. On Windows systems, however, device-specific USB drivers may be required for proper detection.

If a connected device is not recognized, consider the following:

- Install the appropriate USB drivers provided by the device manufacturer
- Use a different USB port on the workstation
- Avoid using USB hubs, as they may introduce instability

In addition, the quality of the USB cable plays a critical role. Low-quality or damaged cables can result in intermittent connectivity, which may disrupt communication or interrupt acquisition processes. It is recommended to use known, reliable cables when working with evidentiary devices.

Pre-connection validation

Before connecting an evidentiary device, it is advisable to validate the setup using a non-evidentiary test device. This step helps confirm that the workstation, tools, and connectivity are functioning as expected.

To perform validation, follow these steps:

1. Connect a test Android device to the workstation
2. Execute the `adb devices` command
3. Confirm that the device is detected and listed correctly
4. Disconnect and repeat the process if necessary to ensure consistency

This practice reduces uncertainty and allows you to become familiar with expected outputs and device behavior before handling actual evidence.

Environment readiness checklist

Before proceeding further, the following conditions should be verified:

- Android platform tools are installed and operational
- ADB commands execute without errors
- Devices can be detected reliably
- A stable USB connection is available
- No background processes are likely to interfere with device interaction

Once these conditions are satisfied, the workstation can be considered ready for forensic use. At this stage, the focus shifts from the environment to the device itself, where careful handling becomes critical to preserving its state and preventing unintended changes.

Handling and preparing the device for examination

Before attempting any form of data extraction, the way an Android device is handled during the initial stages of an investigation can significantly impact the outcome. Improper handling may lead to data modification, loss of volatile information, or even remote wiping of the device. As a result, careful preparation and controlled interaction with the device are essential.

This section focuses on practical steps that should be taken immediately after obtaining access to a device, particularly when the device is powered on or unlocked.

Initial device assessment

The first step is to assess the current condition of the device. This includes determining whether the device is powered on, whether it is locked or unlocked, and whether it is actively connected to any networks.

An unlocked device presents a valuable opportunity. At this stage, you may be able to verify or establish ADB access, assess the device's encryption and lock state, preserve the current unlocked state, and determine which acquisition methods are available. Missing this window can significantly limit the available acquisition options later. These considerations are discussed in more detail in the *Screen lock and access considerations* section.

Network isolation

Modern Android devices are tightly integrated with cloud services and remote management platforms. These services can be used to remotely lock or wipe the device, posing a direct risk to the preservation of evidence. To mitigate this risk, the device should be isolated from all network connections as early as possible.

One approach is to enable airplane mode when interaction with the device is justified. However, Wi-Fi and Bluetooth should be verified separately, as their behavior in airplane mode may vary depending on the device and configuration.Where interaction with the device should be minimized, external isolation techniques may be preferable. The SIM card may also be removed if required, if doing so does not introduce additional risk. As an additional precaution, Wi-Fi and Bluetooth can be manually verified as disabled.

Early network isolation is one of the most critical actions in mobile forensics, as it prevents external interference with the device during examination.

Maintaining device state

If the device is unlocked, steps should be taken to ensure that it remains accessible throughout the examination. Allowing the device to lock may restrict access to protected data, particularly in systems that use file-based encryption.

Practical measures may include increasing the screen timeout duration, enabling options such as 'Stay Awake' within Developer options, and connecting the device to a stable power source. Depending on the device and Android version, additional settings may also need to be considered. For example, on supported Samsung devices, Auto Blocker may restrict USB-based access and should be assessed before acquisition. Android's Inactivity Reboot feature, where present, can automatically reboot a device after an extended period of inactivity, potentially returning it to a Before First Unlock (BFU) state. Similarly, settings that block USB connections while the device is locked may interfere with continued access.

Any changes to these settings should be made only when justified by the acquisition strategy and must be documented. These measures help preserve the device's accessible state and reduce the risk of losing access during the investigation.

Enabling USB debugging

If the device is unlocked and accessible, enabling **USB debugging** may provide an important path for establishing ADB access. However, enabling this setting modifies the device state and should only be performed when justified by the acquisition strategy and properly documented. Once enabled and authorized, USB debugging allows the device to communicate with the forensic workstation via ADB, enabling command execution and controlled data access.

To enable USB debugging, follow these steps:

- Navigate to **Settings** | **About phone**
- Tap **Build number** seven times to enable **Developer options**
- Go back to **Settings** | **Developer options**
- Enable **USB debugging**

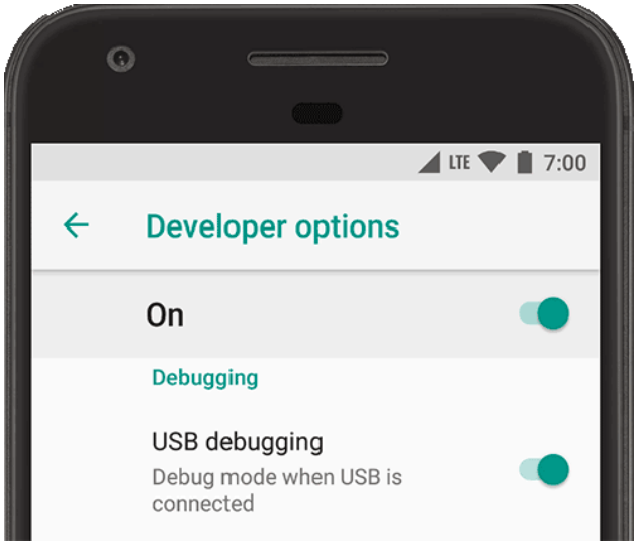


Figure 8.5: USB debugging enabled under Developer options

Once enabled, connect the device to the workstation and authorize the connection when prompted.

Thus, when you connect the device to a new workstation via USB to access ADB, you need to first unlock the device and authorize access by pressing **OK** in the confirmation window, as shown in *Figure 8.6*:

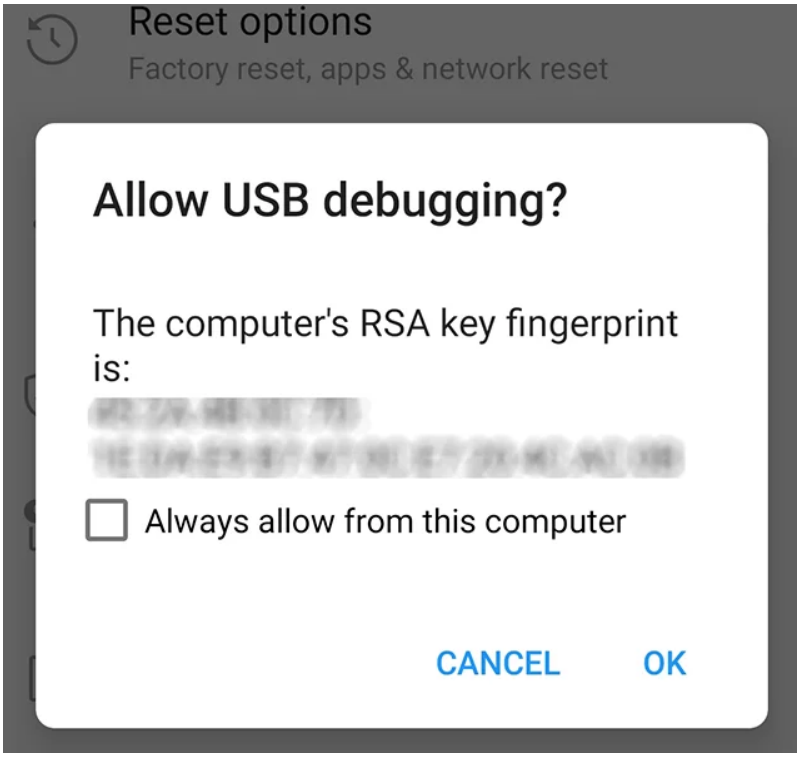


Figure 8.6: Authorizing USB debugging connection

If the **Always allow from this computer** box is checked, the device will not prompt for authorization in the future.

Preserving evidence integrity

While interacting with the device, it is important to minimize changes wherever possible. Every action performed on the device has the potential to alter data, including logs, timestamps, and application states. As a best practice, document all actions performed on the device, avoid installing unnecessary applications and use read-only or minimally invasive techniques whenever possible. In situations where changes are unavoidable, those actions should be clearly recorded to maintain transparency and evidentiary reliability.

In short, the steps taken during this phase can determine the success or failure of the entire investigation. Gaining access to an unlocked device, isolating it from networks, and enabling debugging capabilities can significantly expand the range of acquisition techniques available in later stages. Conversely, failing to secure the device early may result in restricted access, encrypted data, or complete data loss.

Once the device has been secured and its state preserved, the next consideration is whether its screen lock and authentication state will permit access to the data required for examination.

Screen lock and access considerations

One of the most common challenges in Android forensics is dealing with locked devices. In many investigations, the device will be protected by a PIN, password, or biometric mechanism, which directly limits access to user data. Understanding how these protections work, and what options are realistically available, is critical before attempting any acquisition.

Unlike earlier versions of Android, modern devices are designed with strong security controls that make bypassing screen locks significantly more difficult. In many cases, access to meaningful data is only possible when the device is in an unlocked state.

Android devices typically implement one of the following screen lock mechanisms:

- **Pattern lock**: A graphical pattern drawn on a grid
- **PIN**: A numeric code, typically 4 to 6 digits
- **Password**: An alphanumeric passcode
- **Biometric authentication**: Fingerprint or facial recognition

While biometric methods provide convenience, they are typically backed by a PIN or password, which ultimately protects the encrypted data on the device.

Impact on data accessibility

The presence of a screen lock has a direct impact on forensic access. Modern Android devices use file-based encryption, where a significant portion of user data remains encrypted until the device is unlocked after boot. These states are commonly referred to as **before first unlock** (**BFU**) and **after first unlock** (**AFU**). The distinction is important because the device's BFU or AFU state can significantly affect which data is accessible during forensic acquisition. This means the following:

- **Even if storage is accessible, much of the data cannot be decrypted**: Accessing the underlying storage, whether through physical acquisition or low-level extraction, does not guarantee access to usable data. Without the appropriate decryption keys, large portions of user data remain encrypted and cannot be interpreted in a meaningful way.
- **ADB access may be limited or unavailable**: Modern Android devices require explicit user authorization for ADB communication. If the workstation has not previously been authorized and the device cannot be unlocked, the authorization prompt cannot normally be accepted, leaving the connection unauthorized and preventing ADB command execution.

- **Application data under /data remains protected**: Critical forensic artifacts such as application databases, user activity records, and cached content are stored within protected directories. These areas remain inaccessible in a usable form until the device is unlocked and the relevant encryption keys are made available.
- **System and application artifacts may not be fully available**: Certain system services and applications rely on decrypted storage to function. Thus, some artifacts may not be generated, updated, or accessible until the device has been unlocked after boot.

As a result, the ability to access an unlocked device is often the most critical factor in determining the success of an investigation.

Unlocked vs. locked state

From a forensic standpoint, the distinction between an unlocked and a locked device is crucial.

If the device is unlocked at the time of seizure, the following can happen:

- **USB debugging can be enabled**: This allows the device to communicate with a forensic workstation and enables controlled interaction using tools such as ADB.
- **ADB authorization can be granted**: Once authorized, the workstation is trusted by the device, allowing command execution, file access, and data retrieval without further user interaction.
- **Data acquisition options may expand significantly:** After the device has been unlocked, additional credential-encrypted data becomes available to the operating system. Depending on the device, authorization state, and privileges available to the examiner, this may enable acquisition techniques that would otherwise be restricted.
- **More protected data may become accessible:** Once the device has been unlocked after boot, additional encryption keys become available, allowing the operating system and authorized processes to access a broader range of credential-encrypted data. Application sandboxing, SELinux, and other access controls continue to apply.

If the device is locked, it can cause the following:

- **Access to protected data is restricted**: A significant portion of user data remains encrypted and inaccessible until the device is unlocked using the correct credential
- **ADB may remain in an unauthorized state**: Without user interaction on the device, ADB authorization cannot be completed, preventing further communication with the workstation

- **Many logical acquisition methods may fail**: Techniques that rely on system-level access or application data retrieval are often ineffective when the device is locked
- **Opportunities for access become limited**: Without prior authorization or an unlocked state, you may need to rely on alternative or more advanced acquisition methods

Tip

Do not treat a locked device as automatically inaccessible. Before considering bypass techniques, check whether an existing trusted relationship, such as previously authorized ADB access or a paired workstation, provides a less intrusive path to relevant data.

This reinforces the importance of immediate action during initial handling, as discussed in the previous section. Understanding these access limitations is only part of the challenge. In practice, you must apply this knowledge to the specific state and circumstances in which a device is encountered.

Practical and forensic considerations during acquisition

In practice, you rarely have control over the state in which a device is encountered. The device may be powered on or off, locked or unlocked, and connected to external services. As a result, decisions must be made quickly and carefully, based on the condition of the device at the time of acquisition.

If the device is unlocked, priority should be given to preserving that state. Steps such as enabling USB debugging, authorizing ADB access, and preventing the device from locking can significantly improve the chances of successful data extraction. As discussed in the previous section, maintaining device state is often more valuable than attempting complex bypass techniques later.

If the device is locked, the situation becomes more constrained. While various techniques for bypassing screen locks have been documented historically, many of these methods are no longer effective on modern Android versions due to strengthened security controls. Additionally, some bypass techniques may alter the device. For example, a technique that requires booting a custom recovery environment, modifying the boot process, or exploiting the device to obtain elevated privileges may change system state, logs, or other data on the device. Such changes do not automatically invalidate the acquisition, but they must be understood, minimized where possible, and carefully documented. Any bypass technique should therefore be evaluated against its potential impact on evidence integrity and the legal and procedural requirements of the investigation.

It is also important to understand that bypassing the screen lock does not always guarantee access to decrypted data. In devices that use file-based encryption, data may remain inaccessible until the correct credentials have been used to unlock the device after boot. This reinforces the idea that screen locks are part of a broader security model rather than isolated barriers.

In situations where access cannot be obtained through software-based methods, alternative approaches may need to be considered. These may include examining external data sources such as cloud backups, associated accounts, or previously acquired device images. The ability to adapt strategy based on device state is a key skill in modern mobile forensics.

Screen locks should be understood not simply as obstacles to be bypassed, but as part of the device's broader encryption and authentication architecture. Their state can indicate when particular classes of protected data become accessible and which acquisition techniques may be viable. In modern Android environments, the effectiveness of an investigation often depends less on defeating security mechanisms and more on working within their constraints.

A clear understanding of device state, encryption behavior, and access limitations allows you to make informed decisions and select appropriate acquisition techniques. In many cases, success is determined during the initial handling phase, where opportunities to preserve access may be brief but highly consequential.

Access strategies and practical approaches

Once the device has been prepared and its state assessed, the next step is to determine how access can be established in a controlled and forensically sound manner. Unlike earlier Android versions, now there is no single universal approach. The choice of method depends entirely on the device state, particularly whether the device is unlocked and whether debugging access is available. In practice, **access strategies** should be approached in a structured manner, starting with the least intrusive techniques and progressing only when necessary.

If the device is unlocked and USB debugging has already been enabled and authorized, ADB provides the most direct and controlled method of interaction. At this stage, the objective is not to explore the device extensively, but to confirm access and prepare for acquisition.

You should first verify connectivity using `adb devices` command. A device listed in the device state confirms that communication has been established successfully. The shell can then be accessed using `adb shell`. This provides an interactive command-line interface to the Android device and allows you to execute Linux-based commands directly on the device and observe system behavior in real time.

Common initial checks performed through the shell include identifying the current user context, listing accessible directories, and verifying the presence of expected system paths as shown in *Figure 8.7*:

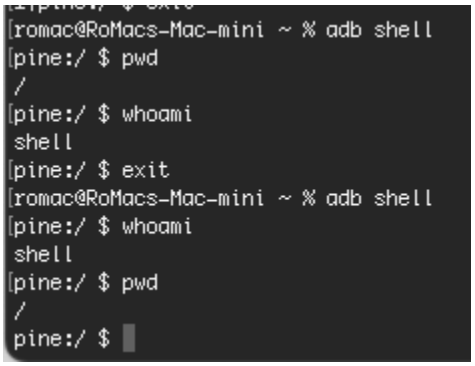


Figure 8.7: Successful communication using adb shell

These commands help confirm the level of access available and provide a baseline understanding of the device environment. However, it is important to understand that shell operates with limited privileges on non-rooted devices. As a result, access to sensitive locations such as `/data` and application-specific directories is restricted. The shell should therefore be used primarily for validation, controlled exploration, and preparation for acquisition rather than deep data extraction.

Even when ADB access is available, modern Android security models impose strict boundaries. Access to application data and protected directories remains restricted without elevated privileges, and many areas of interest will not be accessible in a usable form. As a result, ADB should be viewed as a controlled access channel rather than a complete solution. Its primary value lies in enabling structured interaction and supporting acquisition workflows, rather than providing unrestricted visibility into the device.

The following scenarios represent the most common situations encountered during Android forensic examinations.

Scenario 1: Device locked and no prior access

This is one of the most restrictive scenarios. The device is locked, USB debugging is disabled or not authorized, and no prior trust relationship exists with the workstation. In this state, ADB access is typically unavailable, user data remains encrypted, and interaction with the device is minimal. You should avoid unnecessary interaction that may alter the device. Instead, the focus should shift to preserving the device state and identifying alternative acquisition paths.

If the bootloader is ***locked***, boot-level access is restricted, flashing or booting custom images is not possible, and access is largely limited to non-invasive techniques. If the bootloader is already ***unlocked***, some devices may permit a compatible custom image to be booted or flashed, potentially providing additional access. However, the feasibility and forensic impact are device-specific, and such actions may modify device state. They should therefore be carefully evaluated and documented before use.

On modern devices, generic software-based lock-screen bypass techniques are rarely viable. If device access cannot be established, alternative evidence sources such as cloud data, synchronized systems, backups, and external storage should be considered where legally authorized and available. This scenario emphasizes the importance of preservation over intervention.

Scenario 2: Device locked but ADB previously authorized

In some cases, the device may be locked, but USB debugging was enabled earlier, and the workstation has already been authorized. In this state, the device may appear as device in `adb devices` and ADB commands can still be executed without user interaction. This may provide an opportunity to communicate with the device through ADB and acquire accessible data without first bypassing the screen lock. Even though the device is locked, certain commands and data retrieval operations may still be possible and logs, system information, and limited data may be accessible.

However, access to encrypted user data may still be restricted depending on whether the device has been unlocked since boot. You should confirm ADB access immediately, avoid actions that may cause the device to reboot, and proceed with controlled acquisition techniques. This scenario is highly favorable compared to a fully locked and unauthorized device.

Scenario 3: Device unlocked at time of seizure

This is one of the most advantageous scenarios from a forensic perspective. In this state, the device has already been unlocked after boot, making credential-encrypted user data substantially more accessible. However, application sandboxing, SELinux, profile boundaries, and other security controls may continue to restrict access to specific data. You should act carefully to preserve this state.

Once access is established, logical and file system-level acquisition techniques become viable and a broader range of artifacts can be accessed in decrypted form. This scenario provides the highest probability of successful data extraction and should be prioritized whenever encountered.

Scenario 4: Device powered off

When a device is powered off, encryption keys required to access credential-encrypted user data are not available to the running operating system. After the device boots, it enters the BFU state, in which credential-encrypted data remains inaccessible until the user successfully authenticates.

You must carefully decide whether to power on the device. If the credentials are unknown, powering on may not improve access and could introduce risks such as triggering security mechanisms and enabling remote wipe (if not isolated). In such cases, preserving the device and exploring alternative acquisition methods may be more appropriate.

Scenario 5: Device unlocked but restricted access

In some situations, the device may be unlocked, but access is still limited due to factors such as USB debugging being disabled, Enterprise or mobile device management (MDM) restrictions, or scoped storage and application sandboxing. Even in this state, the device should be stabilized (prevent locking), USB debugging should be enabled if possible, and ADB authorization should be established. If restrictions persist, you may need to rely on user-accessible data like shared storage, explore application-level exports, or consider alternative acquisition techniques.

These scenarios demonstrate that the device state at the time of examination can significantly influence the acquisition options available. An unlocked and accessible device generally provides more opportunities, while a locked, rebooted, or restricted device may require alternative approaches. When standard acquisition methods cannot provide access to the required data, deeper levels of system access may need to be considered. The next section examines one such approach: rooting and privilege escalation.

Rooting and privilege escalation

In Android forensics, there are situations where standard access methods are insufficient to retrieve the data required for an investigation. Many artifacts of interest, particularly application data stored within protected directories, remain inaccessible without elevated privileges. In such cases, you may consider techniques that provide deeper levels of system access, commonly referred to as **rooting** or **privilege escalation**.

In Unix-like operating systems, including Linux, the root account represents the highest level of privilege. A root user has unrestricted control over the system, including the ability to access any file, modify system configurations, and manage processes. Although Android is built on the Linux kernel, direct access to the root account is intentionally restricted in standard devices. This restriction is a core part of Android's security model. Applications and system components operate within isolated environments (sandboxes), preventing them from

accessing data belonging to other applications or interacting with sensitive system areas. Rooting an Android device effectively removes these restrictions by granting superuser privileges.

From a forensic perspective, this expanded access can enable the following:

- Access to protected directories such as `/data`, including application-specific data
- Retrieval of databases, configuration files, and user artifacts not available through standard interfaces
- Interaction with system-level components and services
- More complete file system-level acquisition

However, it is important to understand that rooting is not a neutral operation. It modifies the device state and may impact the integrity and admissibility of evidence. As a result, it must be approached cautiously and only when justified.

Understanding the rooting process

Android device manufacturers implement multiple layers of security to prevent unauthorized access to privileged system components. These protections are designed to restrict direct access to the root account and preserve the integrity of the operating system. Despite these safeguards, various techniques have been developed over time to obtain elevated privileges on Android devices.

The process of rooting differs across devices and manufacturers, but at a conceptual level, it involves gaining elevated access and enabling persistent superuser capabilities. Historically, this was often achieved by exploiting vulnerabilities in the device's firmware or software stack, followed by placing a **superuser binary** (commonly known as **su**) within the system execution path (for example, `/system/xbin/`) and assigning it appropriate permissions. This allowed processes to execute commands with elevated privileges when required.

To better understand how rooting is achieved, we will examine the underlying structure of an Android device. The system is divided into multiple partitions, each responsible for a specific function. These partitions collectively support the device's boot process and runtime environment.

The boot sequence begins with the **bootloader**, which is the first component executed when the device is powered on. Its primary role is to initialize the hardware and load the operating system. The bootloader also enforces security policies by verifying that only trusted and properly signed components are executed. Access to the bootloader menu is typically achieved through a device-specific key combination during startup, allowing the user to boot into different modes, including recovery.

The **recovery partition** is a dedicated environment used for maintenance operations such as installing system updates or performing factory resets. Under normal conditions, this environment accepts only official, signed updates provided by the device manufacturer. This restriction ensures that unauthorized modifications to the operating system cannot be easily introduced. However, if these restrictions are bypassed—for example, by unlocking the bootloader—it becomes possible to replace the standard recovery environment with a modified version.

Custom recovery environments provide extended functionality beyond the default capabilities, including the ability to install unsigned packages, access additional partitions, and facilitate rooting procedures. These modified environments are commonly used as an entry point for gaining elevated access on the device.

Figure 8.8 illustrates the difference between a standard Android recovery environment and a modified recovery environment that provides additional functionality:

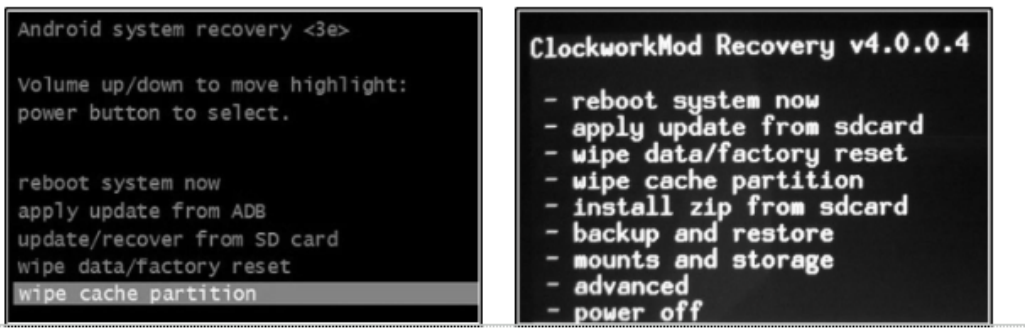


Figure 8.8: Standard vs. modified recovery mode

Earlier rooting approaches often relied on flashing such custom recovery images, followed by installing packages that enable superuser access. Over time, tools have been developed to automate much of this process, making it possible to achieve root access with minimal manual intervention on supported devices. In modern Android systems, however, these techniques have become increasingly difficult to apply.

Starting with Android 7.x, Google introduced and strictly enforced **Verified Boot**, a security mechanism that ensures the integrity of the device during the boot process. Verified Boot establishes a chain of trust in which each stage verifies the authenticity of the next before execution. If any component has been modified or tampered with, the verification process fails, and the device may refuse to boot or enter a restricted state.

This mechanism significantly limits the ability to modify system partitions without detection, making traditional rooting approaches more challenging. Earlier versions, such as Android 6

(Marshmallow), introduced warnings related to system integrity, but from Android 7 onward, these protections became mandatory and more robust.

From a forensic perspective, understanding the rooting process is important not because it is always applied, but because it provides insight into how elevated access can be achieved, what system components are involved, and how modern security mechanisms influence the feasibility and impact of such techniques.

Rooting in modern Android environments

Modern Android devices do not follow a single, universal rooting method. The feasibility and approach depend on factors such as the device manufacturer, chipset, Android version, and security configuration.

A critical factor to consider before attempting to root a device is the state of the bootloader:

- If the bootloader is locked, modification of system components is restricted, and rooting is often impractical without device-specific exploits
- If the bootloader is unlocked, it may be possible to boot or flash modified images that enable elevated access

Unlocking the bootloader is therefore commonly a prerequisite for rooting. This process is typically performed using `fastboot` or vendor-specific procedures and may require enabling OEM unlocking within device settings. Importantly, bootloader unlocking commonly triggers a factory reset on modern Android devices. As an examiner, you should therefore never assume that unlocking a bootloader is a safe means of obtaining forensic access. The behavior of the specific device must be established before any such action is attempted.

Modern rooting techniques in Android have evolved significantly. Earlier methods typically involved directly modifying system partitions and placing binaries such as `su` within protected directories. These changes were persistent and often easy to detect. In contrast, modern approaches rely on what is commonly referred to as **systemless rooting**. Instead of modifying core system files, elevated access is enabled in a way that avoids direct changes to the system partition. A widely used framework for this approach is **Magisk**.

At a high level, Magisk provides root access by patching boot-related components rather than directly modifying the system partition. The exact component involved can vary with the device and Android implementation. By introducing its functionality during the boot process, Magisk can provide controlled superuser access once the system is running.

Figure 8.9 shows an example of a device configured with Magisk for root access:

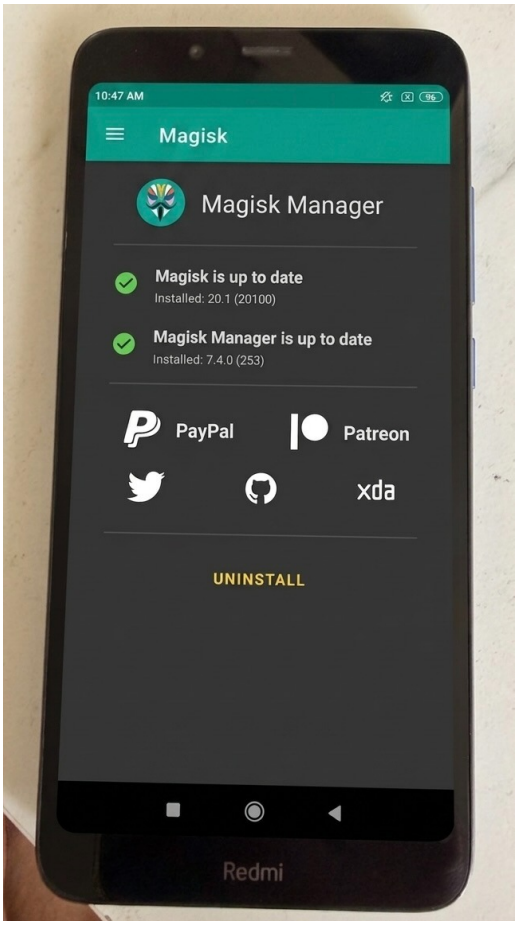


Figure 8.9: Android device rooted using Magisk

This model introduces several important differences compared to traditional rooting:

- The system partition remains unchanged, reducing obvious signs of modification
- Root access is enabled dynamically at runtime rather than permanently embedded
- Elevated privileges are typically controlled through a management interface, where access can be granted or denied

Traditional rooting methods often leave clear artifacts within system directories, making them relatively easy to identify. Systemless approaches, however, may leave fewer visible traces in the filesystem. Instead, indicators of root access may be found in areas such as modified boot images, active processes, or root management frameworks.

In practical terms, a modern rooting workflow on a compatible device may involve unlocking the bootloader (if permitted), booting or flashing a modified boot image, and enabling controlled elevated access through a management interface.

Despite these advancements, rooting is not universally achievable. Modern Android devices incorporate strong security mechanisms that may prevent or detect such modifications. As a result, the feasibility of rooting must always be evaluated in the context of the specific device, its security configuration, and the objectives of the investigation. To see how this difference affects access in practice, the next section demonstrates ADB shell access on non-rooted and rooted Android devices.

Root access: ADB shell

A normal Android phone does not allow you to access certain directories and files on the device. For example, try to access `/data/data folder` on an Android device that is not rooted. You will see the following message on a Windows workstation:

```
C:\android-sdk\platform-tools>adb.exe shell
shell@android:/ $ cd /data/data
shell@android:/data/data $ ls
opendir failed, Permission denied
255|shell@android:/data/data $
```

Figure 8.10: Permission denial without root access

On a rooted device, first establish an ADB shell using the `adb shell` command. *Figure 8.11* shows the output of this command:

```
C:\android-sdk\platform-tools>adb.exe shell
root@android:/ # cd /data/data
root@android:/data/data # ls
android.googleSearch.googleSearchWidget
com.android.MtpApplication
com.android.Preconfig
com.android.apps.tag
com.android.backupconfirm
com.android.bluetooth
com.android.browser
```

Figure 8.11: Successful execution of shell command on a rooted device

Depending on how root access is implemented, the resulting shell may still run as the shell user. In environments such as Magisk, you can then issue the `su` command to request root privileges, as shown in *Figure 8.12*. Notice that the prompt typically changes from `$` to `#` after elevated privileges are obtained:

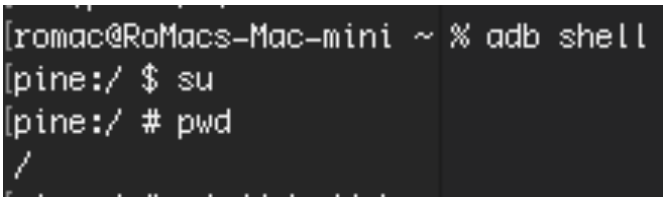


Figure 8.12: Running as root on Magisk rooted devices

Thus, rooting a phone enables you to access folders and data that are otherwise not accessible. Also note that `#` symbolizes root or superuser access, while `$` reflects a normal user, as shown in the preceding command lines.

Limitations in modern Android

While rooting was once a commonly used technique in Android forensics, advances in platform security have significantly reduced its effectiveness in modern devices. Contemporary Android systems incorporate multiple layers of protection that are specifically designed to prevent unauthorized modification and restrict elevated access.

Most devices now ship with bootloaders locked by default, preventing changes to critical system components. Mechanisms such as verified boot enforce a strict chain of trust, ensuring that only authentic and unmodified software is executed during startup. In addition, sensitive data is protected using hardware-backed key storage, which isolates encryption keys from the operating system and makes them resistant to extraction, even in the presence of elevated privileges.

Modern devices are also capable of detecting system modifications. Attempts to alter system partitions or bypass security controls may result in restricted functionality, warning states, or, in some cases, data being rendered inaccessible. It is therefore important to recognize that obtaining root access does not necessarily equate to full data access. If a device has not been unlocked and the required decryption keys are not available, significant portions of user data may remain encrypted and inaccessible.

Note

Root access and decrypted data access are not the same thing. Root privileges may allow an examiner to bypass filesystem permission boundaries, but they do not automatically make encryption keys available. The device's lock state and encryption state remain critical factors in determining what data can be acquired.

Forensic risks and considerations

Rooting introduces direct risks to evidence integrity. The process may modify system files, alter timestamps, or introduce new artifacts that affect subsequent analysis. Key considerations include the following:

- **Justification**: Rooting should only be performed when less intrusive methods are insufficient
- **Documentation**: All actions must be recorded in detail
- **Validation**: The impact on data integrity should be assessed and, where possible, verified
- **Legal compliance**: Actions must align with investigative and legal requirements

In most forensic workflows, rooting is treated as a last-resort technique.

Tip

When possible, test an intrusive procedure on an equivalent device before applying it to evidence. A device with the same model, Android version, security patch level, and bootloader configuration can help reveal whether a procedure causes a reset, modifies data, or behaves differently than expected.

Practical and forensic perspectives

The role of rooting in Android forensics has evolved significantly. Rather than being a default approach, it is now one of several techniques that must be carefully evaluated based on the device, its security configuration, and the objectives of the investigation. In many cases, alternative methods such as logical acquisition, cloud-based data retrieval, or analysis of synchronized systems can provide sufficient evidence without altering the device. In others, rooting may still be necessary to access critical data that cannot be obtained through less intrusive means.

Rooting and privilege escalation offer deeper visibility into the device, but they are inherently intrusive and may impact the integrity of the evidence. As a result, the decision to pursue root access must be guided by a clear understanding of both its benefits and its risks. In modern Android forensics, success is not defined by the ability to obtain root access, but by the ability to select the most appropriate method for the given scenario while maintaining the reliability and defensibility of the evidence. With these considerations in mind, the next section shifts from root access to the practical use of ADB shell commands for navigating and examining an Android device.

Working with ADB shell: Essential Linux commands

When using `adb shell`, you interact with the Android device through a Linux-based command-line interface. The following are some commonly used commands that help in navigating and understanding the device during examination:

- **ls**: The `ls` command lists the files and directories present in the current directory

```
romac@RoMacs-Mac-mini ~ % adb shell
pine:/ $ su
pine:/ # ls
acct                init.miui.early_boot.sh              odm
bin                 init.miui.google_revenue_share.rc    oem
bugreports          init.miui.google_revenue_share_v2.rc persist
cache               init.miui.google_revenue_share_vdf.rc proc
charger             init.miui.nativedebug.rc             product
config              init.miui.post_boot.sh               res
cust                init.miui.rc                         sbin
d                   init.rc                              sdcard
data                init.recovery.hardware.rc            storage
default.prop        init.recovery.qcom.rc                sys
dev                 init.usb.configfs.rc                 system
etc                 init.usb.rc                          ueventd.rc
init                init.zygote32.rc                     vendor
init.carrier.rc     lost+found                           verity_key
init.environ.rc     metadata
```

Figure 8.13: ls command to see files and folders

- **cat**: The `cat` command reads one or more files and prints them to standard output, as shown in the following command lines. For example, in the previous scenario, to read `default.prop` file, `cat` command can be used as follows:

```
pine:/ # cat default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.miui.has_cust_partition=true
ro.appsflyer.preinstall.path=/data/miui/pai/pre_install.appsflyer
ro.miui.pai.preinstall.path=/data/miui/pai/
ro.secureboot.devicelock=1
ro.booking.channel.path=/cust/etc/.booking.data.aid
ro.actionable_compatible_property.enabled=true
ro.secure=1
security.perf_harden=1
ro.adb.secure=1
ro.allow.mock.location=0
ro.debuggable=0
ro.opa.eligible_device=true
ro.setupwizard.mode=OPTIONAL
setupwizard.feature.predeferred_enabled=false
```

Figure 8.14: cat command to view files

- **cd**: The `cd` command is used to change from one directory to another. This is more frequently used while navigating from one folder to another. The following example shows the commands used to change to the system folder:

```
pine:/ # cd /system
pine:/system #
```

Figure 8.15: cd command to switch directories

Now, let's look at some other key commands that are worth noting:

- **cp**: The `cp` command can be used to copy a file from one location to another. The syntax for this command is: `$ cp [options] <source><destination>`.
- **dd**: The `dd` command is used to copy a file, converting and formatting it according to the operands. On devices where sufficient privileges and block-device access are available, `dd` can be used to copy the contents of a partition or block device into an image file. More details of imaging are covered in *Chapter 4*. The following is the syntax that needs to be used with this command: `dd if=/test/file of=/sdcard/sample.image`.
- **grep**: The `grep` command is used to search files or output for a particular pattern. The syntax for this command is: `grep [options] pattern [files]`.

- **pwd**: The `pwd` command displays the current working directory. The syntax for this command is: `pwd`.
- **id:** Displays the current user and group identifiers. This is useful for determining the privilege context of the ADB shell.
- **getprop:** Displays Android system properties. It can be used to obtain information such as device model, Android version, build details, and other system configuration values.
- **mount:** Displays mounted filesystems and their mount points and options, helping you understand the device's current storage layout.
- **exit**: The exit command can be used to exit the shell you are in. Just type `exit`.

Tip

Capture `getprop` output early in the examination. It provides a useful snapshot of the device configuration and can help identify the exact build, security patchlevel, and other properties needed when evaluating device-specific acquisition techniques.

These commands provide the basic tools needed to navigate and examine an Android device through the ADB shell. Before using these capabilities for acquisition, however, the device and forensic environment should be validated to ensure that extraction can proceed safely and reliably.

Validation and pre-extraction checklist

Before initiating any form of data acquisition, it is essential to validate the current state of the device and the forensic environment. Actions taken at this stage directly influence the success of extraction and the integrity of the evidence. A structured, checklist-driven approach helps ensure that critical steps are not overlooked and that the examination proceeds in a controlled and defensible manner.

This stage serves as a final checkpoint between preparation and acquisition. Before proceeding with data extraction, you should confirm that the following prerequisites have been satisfied:

1. **Device state verification**: The first step is to confirm the current state of the device. This includes determining whether the device is powered on, locked or unlocked, and whether it has been recently rebooted. An unlocked device significantly increases the range of available acquisition techniques, particularly in environments where encryption is enforced. If the device is locked, access to meaningful data may be restricted, and alternative strategies may need to be considered.

2. **Network isolation confirmation**: The device should be verified to be fully isolated from all network connections. This includes cellular networks, Wi-Fi, and Bluetooth. Failure to isolate the device may expose it to risks such as remote wiping, remote locking, or synchronization events that could alter existing data. Even if airplane mode has been enabled, additional checks should be performed to ensure that all communication channels are disabled.
3. **ADB connectivity and authorization**: If ADB is being used as part of the acquisition workflow, connectivity should be verified prior to extraction. This includes confirming the following:
    - The device is detected by the workstation
    - USB debugging is enabled
    - The workstation has been authorized on the device

    A device listed as unauthorized or offline indicates that further interaction will not be possible until the issue is resolved. Verifying ADB connectivity at this stage avoids interruptions during acquisition.
4. **Access level assessment**: The level of access available on the device should be clearly identified. This includes determining whether you have the following access:
    - Standard user-level access
    - ADB shell access
    - Elevated (root) privileges

    Understanding the available access level is critical in selecting an appropriate acquisition method. Attempting techniques that exceed the current access level may result in failure or unintended modifications.
5. **Acquisition strategy alignment**: Based on the device state and access level, you should identify the most appropriate acquisition approach. Not all methods are suitable for every device, and the chosen technique must align with both technical feasibility and investigative requirements. At this stage, you should have a clear plan for how data will be extracted, including any tools, commands, or workflows that will be used.
6. **Documentation and evidence preservation**: Before proceeding, the current state of the device should be documented. This may include the following:
    - Photographing the device screen
    - Recording lock status and visible notifications

- Noting system time and device condition
- Capturing relevant logs or command outputs

Proper documentation ensures transparency and supports the defensibility of the examination process.

Final pre-extraction check

Before initiating extraction, you should confirm the following conditions are met:

- The device state is understood and stable
- Network isolation has been enforced
- Communication with the device is functioning
- The access level is clearly established
- The acquisition method has been selected and validated
- All relevant observations have been documented

Only after these checks have been completed should you proceed with data acquisition. Together, these preparation, and validation steps help ensure that the device is handled in a controlled manner, potential sources of evidence are preserved, and the chosen acquisition approach is appropriate for the device's current state. With these foundations in place, the investigation can move from preparation to the actual extraction of data.

Summary

In this chapter, we established the foundational steps required before performing data acquisition on an Android device. We began by setting up a controlled forensic environment and understanding the role of core tools such as ADB in enabling structured interaction with the device.

We then examined the importance of proper device handling during the initial stages of an investigation. Preserving the device state, isolating it from network connectivity, and identifying opportunities when the device is unlocked were highlighted as critical factors that directly influence the scope of accessible data. The impact of screen locks and modern encryption mechanisms was explored in detail, emphasizing how the device's lock state determines the availability of user data. This led to practical access strategies, where different scenarios were evaluated based on device condition, access level, and technical constraints.

We also discussed rooting and privilege escalation as advanced techniques for obtaining deeper access. While these methods can expand visibility into protected areas of the system, they introduce risks related to data integrity and must be carefully justified and documented. Finally, we introduced a structured validation process to ensure that all prerequisites are satisfied before beginning extraction. This checklist-driven approach reinforces disciplined forensic practice and minimizes the risk of errors during acquisition.

9

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:







Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> **Note**: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 9.1* on desktop), search for this book by name and select the correct edition.

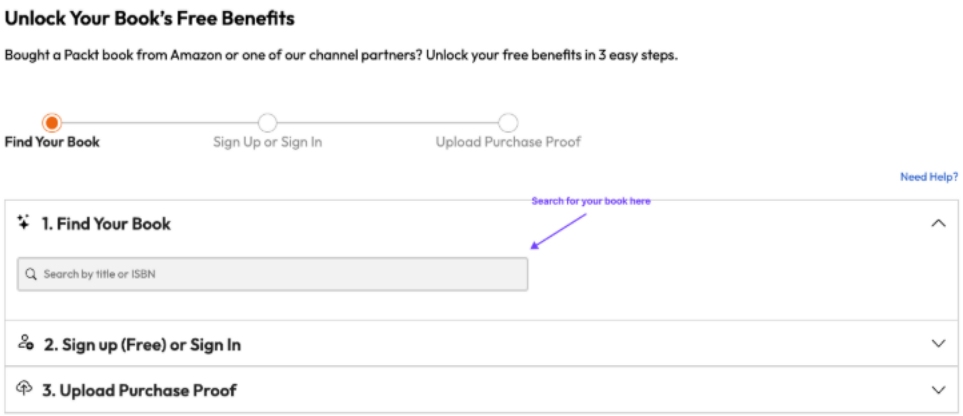


Figure 9.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

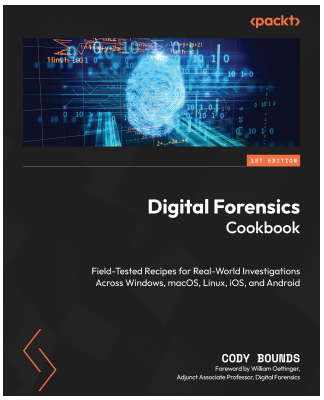
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Digital Forensics Cookbook

Cody Bounds

ISBN: 978-1-80512-755-0

- Perform triage and acquire evidence during live investigations
- Collect artifacts remotely using incident response workflows
- Analyze evidence across Windows, macOS, Linux, iOS, and Android
- Recover deleted data and manually carve evidence when tools fail
- Crack encrypted containers using custom password dictionaries
- Use regex and SQLite queries to uncover hidden investigative clues
- Detect anti-forensic techniques and metadata tampering
- Analyze Windows memory using Volatility to uncover live artifacts



SOC Analyst Career Guide

Kyler Kent

ISBN: 978-1-83546-746-6

- Explore security operations and see how it fits into an overall business
- Discover the key functions and real-world duties of a SOC analyst
- Understand and master the tools of the trade, including SIEM
- Develop a home lab with an operational SIEM and EDR solution
- Create a compelling portfolio to impress prospective employers
- Perfect your interviewing skills and land a SOC analyst job

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Practical Mobile Forensics*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/1806107775

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

ADB device states
- device 227
- offline 228
- unauthorized 227

ADB shell **245, 246**
- working with 248 – 250

Android
- bootloader 213
- chain of trust 212
- encryption 207
- evolution 194
- filesystems 216 – 218
- forensic impact 195
- hardware-backed security mechanisms 209
- storage and filesystem layout 213
- system integrity 212
- Verified Boot 212

Android Keystore **209, 210**

Android Runtime (ART) **198 – 201**

Android Verified Boot (AVB) **212**

Android architecture **196**
- acquisition strategy 220
- Android Runtime (ART) 198
- application framework 199
- hardware abstraction layer (HAL) 198
- Linux kernel layer 198
- native libraries 198, 199
- system applications 199
- user applications 199

Android debug bridge (ADB) **6, 225**
- connectivity, verifying 226

Android devices
- access strategies 237 – 240
- device state, maintaining 231
- evidence integrity, preserving 233
- final pre-extraction check 252
- initial device assessment 230
- network isolation 231
- screen lock and access considerations 234 – 237
- USB debugging, enabling 232, 233
- validation checklist 250 – 252

Android platform tools
- installing 225, 226

Android security model **203**
- application sandbox enforcement 203, 204
- application signing 205, 206
- inter-process communication (IPC) 205
- Linux permission model 203
- permission-based model 204, 205
- Security-Enhanced Linux (SELinux) 206
- trust model 205, 206
- UID/GID 203

Apple Devices **93**

Apple file system (APFS) **41, 42, 70**
- architecture 41
- forensic considerations 45
- iOS directories 43
- snapshots 44
- versus HFS+ 43

Apple mobile file integrity (AMFI) **36**

access strategies **237**

activation lock **63**

advanced data protection (ADP) **31, 88, 108**

after first unlock (AFU) **46, 234**

agent-based acquisition **72, 75, 76**

app sandboxing **37**

application data **215**

application execution environment **200**
Android Runtime (ART) 200, 201
Dalvik Virtual Machine (DVM) 200, 201
application framework **199**
application isolation **195**
application sandbox enforcement **203, 204**
application security, iOS security model **37**
app sandboxing 37
code signing 37, 38
entitlements 38
application storage model **202**

B

Bootloader **24, 213, 241**
backup contents **95**
backup-based evidence **168**
backup-derived artifacts **120**
backups
forensic best practices 97
before first unlock (BFU) **46, 234**
best practices, mobile forensic investigations
evidence, documenting 21, 22
evidence, preserving 21
evidence, securing 20, 21
reporting 22

C

Cellebrite UFED **40, 97**
Chrome time **123**
cat command **248**
cd command **249**
chain of trust **47, 212**
checkra1n **73**
chip-off **15**
code-signing infrastructure **37, 38**
commercial forensic platforms **169, 170**
commercial iOS forensic platforms
activity reconstruction 180
artifacts, filtering 180
artifacts, searching 180
automated artifact parsing 179, 180
critical findings, validating 181
differ in practice 182
evidence, exporting 181
evidence, importing 177
evidence, preparing 178
reports, generating 182
timeline analysis 180
working with 177
content providers **199**
cp command **249**
credential encrypted (CE) storage **207, 208**
custom recovery **242**

D

DB Browser for SQLite (DB4S)
URL 126
Dalvik Virtual Machine (DVM) **200, 201**
Direct Boot model **207**
data execution prevention (DEP) **36**
data partition **213**
dd command **249**
deleted data **158**
device encrypted (DE) storage **207, 208**
device firmware upgrade (DFU) **47, 54, 59**
entering 59, 60
verifying 60
device interaction and behavioral intelligence artifacts
derived media 156
local dictionary 155
media files 155
photos 155
property lists 152, 153
third-party application data 156, 157
thumbnails 156
user's phone numbers 151
wallpaper and personalization artifacts 156

device settings **123**
digital forensics **5**
dynamic partitions **214**

E

EXT4 filesystem **218**
Elcomsoft Phone Breaker **97**
Elcomsoft iOS Forensic Toolkit **40**
emulated filesystems **219**
emulated storage (/sdcard) **215**
encrypted backups **95, 96**
encryption **61**
encryption, Android **207**
credential encrypted (CE) storage 208
device encrypted (DE) storage 208
file-based encryption (FBE) 207
forensic implications 209
full disk encryption (FDE) 207
end-to-end encryption **39**
entitlements **38**
exit command **250**
exploits **61**
external storage **215**

F

F2FS filesystem **218, 219**
Finder **93**
file-based acquisitions
unallocated space 163
file-based encryption (FBE) **46, 207**
filesystem acquisition **70, 89, 168**
filesystem artifacts **119**
filesystem extractions
SQLite databases, locating 131, 132
filesystems, Android **216 – 218**
emulated 219
EXT4 218
F2FS 218, 219
forensic considerations 219
journaling 219
legacy 219
virtual 219
write-ahead logging (WAL) mode 219
forensic integrity **224**
forensic pyramid classification system **13, 14**
chip-off 15
hex dump 14
in-system programming (ISP) 15
logical extraction 14
manual extraction 14
micro read 16
physical extraction 14
forensic report **22**
forensic workstation **225**
drivers and system considerations 229
environment readiness checklist 230
pre-connection validation 229
fqlite
URL 126
full disk encryption (FDE) **46, 207**

G

GID keys **34**
getprop **250**
grep command **249**

H

hardware abstraction layer (HAL) **198**
hardware security, iOS security model **34**
GID keys 34
memory integrity enforcement (MIE) 34
Secure Enclave processor (SEP) 34
UID key 34
hardware-backed security mechanisms, Android **209**
Android Keystore system 209, 210
StrongBox 211

trusted execution environment (TEE) 210, 211

hex dump **14**

hierarchical file system plus (HFS+) **41**

I

iBackup Viewer **97, 101**
iOS backup, analyzing 101 – 105

iCloud backups **90, 91, 105**
accessing 107
acquiring, with Magnet AXIOM 109 – 113
contents 106
impact of ADP 108
working 106

iExplorer **97**

iOS Keychain **38**

iOS Keychain and iCloud security **38**
access control 39
access groups 39
end-to-end encryption 39
extraction challenges 40
forensic analysis 40

iOS acquisition methods
decision framework, for selecting 83 – 85

iOS architecture layers **32**
cocoa touch layer 33
core OS layer 32
core services layer 33
media layer 33

iOS artifact analysis
artifact extraction, automating across multiple backups 184 – 186
automated timeline construction and correlation 186 – 188
automation techniques 182
automation, best practices 189
SQLite artifact extraction, automating 183, 184

iOS backups **88**
analyzing, with iBackup Viewer 101 – 105
extracting, with idevicebackup2 98 – 101
in forensic investigations 88, 89
SQLite files, locating from 132 – 134

iOS data analysis **118**
actual evidence 118, 119
backup-derived artifacts 119, 120
filesystem artifacts 119, 120
parsed results 118, 119
raw data validation 119
tool output 119

iOS database files
analyzing 135
behavioral intelligence artifacts 150, 151
calendar events artifact 144 – 146
call history artifact 140 – 142
contacts artifact 136 – 140
device interaction 150, 151
history 148
messages artifact 143, 144
notes artifact 147, 148
Safari bookmarks 148
tab artifacts 148

iOS devices
operating modes 54 – 60

iOS diagnostic and unified log acquisition **80**
best practices 83
collecting 81
full unified log archives, acquiring 82, 83
limitations 83
manual extraction pathways 82
unified logs, forensic value 81

iOS ecosystem **30**
App Store 31
Apple ID 31
iCloud 31
iOS versions 30
sandboxing 31
Secure Boot chain 31
Secure Enclave 31
supported devices 30

iOS evidence
SQLite databases, locating in 130

iOS filesystem acquisition **70**
agent-based acquisition 72, 75, 76
jailbreak-assisted acquisition 73 – 79
significance 71
iOS forensics
automation, for scaling errors 176
automation, roles and limits 175
candidates for automation 175
human-guided automation 177
interpretation, examiner-driven 176
provenance, maintaining during automation 176
iOS forensics tool validation **171**
chronological integrity 172, 173
cross-tool validation 171, 172
defensibility 174, 175
deleted data recovery 173
documentation 174
parsed output 172
parser limitations 174
record count confirmation 173
residual artifact validation 173
schema interpretation 173
source artifacts 172
timestamp verification 172, 173
version compatibility 174
iOS forensics tooling landscape **168**
acquisition 168
analysis 169
automated and manual analysis, combining 170, 171
commercial forensic platforms 169, 170
functional roles 168, 169
manual analysis 170
manual and open-source tools 170
parsing 169
presentation 169
processing 169
iOS isolation techniques **62, 63**
activation lock 63
battery considerations 64
power considerations 64
remote wipe 63
iOS logical acquisition **64**
best practices 66
commercial forensic tool, using 67 – 70
forensic value 66
in practice 66
limitations 65
prerequisites 64
tools 65
iOS physical acquisition **79, 80**
iOS security model **34**
application security 37
hardware security 34
system security 35
iOS time **120**
iOS-specific legal and forensic considerations **61**
chain of custody 62
documentation 62
encryption 61
exploits 61
jailbreaking 61
legal authority 62
legal boundaries 61
passcodes 61
Secure Enclave 61
warrants 62
iTunes **93**
id **250**
idevicebackup2
iOS backup, extracting 98 – 101
in-system programming (ISP) **15**
info.plist **95**
inter-process communication (IPC)
securing 205

J

jailbreak tools **49**
jailbreak-assisted acquisition **73 – 79**
jailbreaking **48, 61**
forensic implications 48 – 50
hardware-based jailbreaks 49

software-based jailbreaks 49
joint test action group (JTAG) **15**
journaling **219**

K

kernel **36**
kernel protection mechanisms
Apple mobile file integrity (AMFI) 36
data execution prevention (DEP) 36
kernel text read-only region (KTRR) 36
pointer authentication codes (PAC) 36
kernel text read-only region (KTRR) **36**

L

Letos
URL 126
Linux kernel layer **198**
Linux permission model **203**
legacy filesystems **219**
legal boundaries **61**
libimobiledevice **97**
local backups **90, 91**
creating 93
locating 94
lockdown file **64**
logical acquisition **64, 89**
ls command **248**

M

Mac absolute time **122**
Magisk **243**
Magnet AXIOM **40, 109**
iCloud backup, acquiring 109 – 113
manifest.db **95**
manifest.plist **95**
manual validation **119**
media transfer protocol (MTP) **6**
memory integrity enforcement (MIE) **34**
metadata **120**
metadata encryption **208**
micro read **16**
mobile devices
as sources of digital evidence 4, 5
evidence extracted, analyzing 17, 18
evidence extracted, examining 17, 18
evidence extracted, types 16, 17
evidence handling 18 – 20
mobile forensic investigations
best practices 20 – 23
process 6 – 8
mobile forensics **5**
Android 12
iOS 12
principles 5, 6
mobile forensics workflow
acquisition 10
analysis 11
applying 8
archiving 12
documentation 11
evidence intake 8, 9
examination 11
identification 9
isolation 10
legal authority 8, 9
preparation 9
preservation 9
reporting 11
verification 11
modern iOS security
impact 163
modern mobile investigations
challenges 23 – 26
mount **250**

N

native libraries **198, 199**
normal mode **54 – 56**

O

Open Handset Alliance (OHA) **194**
Oxygen Forensic Detective **97**
operating modes, iOS devices **54**
device firmware upgrade (DFU) mode 54, 59, 60
normal mode 54 – 56
recovery mode 54 – 58

P

Physical Analyzer **97**
pairing record **64**
parsed output **172**
parsing **179**
passcodes **61**
permission model **204, 205**
physical acquisition **79, 89**
physical extraction **14**
picture transfer protocol (PTP) **6**
plist files **153 – 155**
plutil utility **152**
pointer authentication codes (PAC) **36**
pre-extraction phase **224**
pre-extraction workflows **224**
privilege escalation **240**
property lists **152**
pwd command **250**

R

raw data validation **119**
recently deleted **163**
recovery loop **58**
recovery mode **54 – 57**
entering 58
exiting 58
recovery partition **242**
rooting **240 – 242**
forensic risks and considerations 247
in modern Android environments 243 – 245
limitations, in modern Android 246
practical and forensic perspectives 247
runtime approach **204**

S

SQLite **124**
evidence, parsing 130
filesystem extraction, versus backup-based SQLite evidence 130
freelist pages 159, 160
key commands 127 – 129
SQLite artifact extraction
automating 183, 184
SQLite command-line client
URL 126
SQLite databases **124**
connecting to 127
examining 125
locating, in filesystem extractions 131, 132
locating, in iOS evidence 130
working with 126
SQLite files
locating, from iOS backups 132 – 134
SQLiteSpy
URL 126
Secure Boot chain **35, 47**
Secure Enclave **24, 34**
Secure Enclave processor (SEP) **46, 61**
Secure iCloud Key Escrow **39**
Security-Enhanced Linux (SELinux) **206**
StrongBox **210, 211**
sandboxing **37**
scoped storage restrictions **215**
security enhancements, on iOS forensics **45**

file system encryption 46
full disk encryption 46
imaging limitations 47
Secure Boot chain 47
Secure Enclave processor (SEP) 46
signing certificate 205
soft-delete mechanism 163
source artifacts 172
sqlite3 tool 126
standard backup metadata files
info.plist 95
manifest.db 95
manifest.plist 95
status.plist 95
status.plist 95
storage and filesystem layout, Android 213
A/B updates 214
application data 215
dynamic partitions 214
emulated storage (/sdcard) 215
general partition structure 213
scoped storage restrictions 215
user artifacts 215
superuser binary 241
sysdiagnose 81
system applications 199
system security, iOS security model 35
kernel integrity 36
Secure Boot chain 35
software updates 35
system-related partitions 213
systemless rooting 243

T

TrustZone 24
test access ports (TAPs) 15
time zones 123
timestamp interpretation 124
tool validation 171
trusted execution environment (TEE) 210, 211
two-character hexadecimal folders 95

U

UID key 34
UID/GID 203
Unix timestamps 121
unencrypted backups 96
unified logging system 80
unified logs 80
user artifacts 215
user-installed applications 199

V

Virtual A/B updates 214
validation 119
virtual filesystems 219

W

WebKit time 123
write-ahead log (WAL) files 161
write-ahead logging (WAL) mode 161, 162, 219