Forensic Science International: Digital Investigation 54 (2025) 301941

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

Decrypting messages: Extracting digital evidence from signal desktop for windows

Gonçalo Paulino [a,*], Miguel Negrão [a,c], Miguel Frade [a,c], Patrício Domingues [a,b]

[a] School of Technology and Management - Polytechnic Institute of Leiria, Leiria, Portugal
[b] Instituto de Telecomunicações, Portugal
[c] Computer Science and Communication Research Centre, Portugal

ARTICLE INFO

Keywords:
Digital forensics
Signal
Windows
Encryption
Electron
Instant messaging

ABSTRACT

With growing concerns over the security and privacy of personal conversations, end-to-end encrypted instant messaging applications have become a key focus of forensic research. This study presents a detailed methodology along with an automated Python script for decrypting and analyzing forensic artifacts from Signal Desktop for Windows. The methodology is divided into two phases: i) decryption of locally stored data and ii) analysis and documentation of forensic artifacts. To ensure data integrity, the proposed approach enables retrieval without launching Signal Desktop, preventing potential alterations. Additionally, a reporting module organizes extracted data for forensic investigators, enhancing usability. Our approach is effective in extracting and analyzing encrypted Signal artifacts, providing a reliable method for forensic investigations.

1. Introduction

Instant messaging applications have been the focus of numerous studies aimed at understanding their functionality and the different artifacts that can be collected for forensic investigations (Son et al., 2022). Due to the nature of these applications, their data have significant forensic value, enabling investigators to infer aspects of interpersonal relationships to an individual's mental state (Steel, 2014; Andriotis et al., 2014).

With the increasing concern of everyday users about the security and privacy of their personal conversations (Madden, 2014; Turner et al., 2019), the number of instant messaging solutions that market themselves as secure has grown. These solutions incorporate various security mechanisms — such as end-to-end encryption (E2EE), self-destructing messages, local storage encryption, and forward secrecy — which introduce new challenges to the forensic analysis of these applications. Note that self-destructing messages, often referred to as ephemeral messages, have attracted significant interest within the digital forensic scientific community (Abegaz et al., 2024; Heath et al., 2023).

Table 1
List of Acronyms.

Acronym	Meaning
ACI	Account Identifier
AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining
CSV	Comma Separated Values
DPAPI	Data Protection API
GCM	Galois/Counter Mode
PNI	Phone Number Identifier
UTC	Universal Time Coordinated
UUID	Universally Unique Identifier
UWP	Universal Windows Platform

Notable examples of privacy-focused instant messaging applications include centralized solutions such as Signal,[1] WhatsApp,[2] Threema,[3]

* Corresponding author.
E-mail address: 2230469@my.ipleiria.pt (G. Paulino).

[1] https://signal.org.
[2] https://www.whatsapp.com.

https://doi.org/10.1016/j.fsidi.2025.301941
Received 11 April 2025; Received in revised form 24 May 2025; Accepted 1 June 2025
Available online 10 June 2025

and Wire,[4] along decentralized alternatives such as Briar,[5] Session,[6] and SimpleX Chat.[7] Among these, Signal is one of the most widely adopted and pioneered the Signal Protocol, which has also been adopted by WhatsApp (Marlinspike, 2016) — the leading messaging app — making it an ideal candidate for this study. At the time of this writing, Signal has surpassed 100 million downloads on the Google Play Store[8] and 850000 ratings on the Apple Store.[9]

Signal is an open-source instant messaging application that offers robust privacy and security features (Cohn-Gordon et al., 2020). Its centerpiece is end-to-end encryption, powered by the Signal Protocol, an open source cryptographic protocol developed internally by the Signal Foundation (Signal Foundation, 2025c).

The Signal Protocol combines multiple algorithms, including Double Ratchet (Perrin and Marlinspike, 2016), Curve25519 (Langley et al., 2016), and AES-256 (NIST, 2001), to ensure the security of user communications, guaranteeing not only End-to-End Encryption, but also forward and future secrecy (Signal Foundation, 2025c; Kret and Schmidt, 2024).

Signal offers Android[8] and iOS[9] clients for mobile devices, as well as an Electron client for desktop environments — Windows, macOS, and Linux. This paper focuses on the Electron client for the Windows operating system, which provides users with the same standalone messaging functionalities as the mobile apps; however, Signal Desktop requires an initial connection to a mobile device for account creation and synchronization. Our research spanned Signal Desktop[10] versions **7.32.0** to **7.40.1**, released between November 2024 and February 2025.

This study aims to address the forensic challenges posed by Signal Desktop's encrypted local data on Windows. Specifically, we set out to:

1. Investigate how Signal Desktop stores and secures forensic artifacts locally.
2. Identify and document the cryptographic mechanisms involved in securing those artifacts.
3. Develop and validate a methodology for decrypting the locally stored data without launching the application.
4. Analyze the decrypted artifacts to determine their forensic value;
5. Create an automated tool that streamlines both the decryption and reporting processes.

This study provides several key contributions to the field of digital forensics, specifically focusing on Signal Desktop's forensic artifacts:

i) **Documentation of Signal Desktop's Decryption Process:** We document the step-by-step process required to decrypt Signal Desktop's locally stored data, including the retrieval of cryptographic materials and the use of SQLCipher and Electron's `safeStorage` API. This fills a gap in the literature with respect to the forensic analysis of this application.

ii) **Development of an Automated Forensic Tool:** We introduce `SignalForensics`— https://github.com/labcif/SignalForensics — a Python-based tool that automates the decryption and extraction of forensic artifacts from Signal Desktop's local files. This tool eliminates the need for manual decryption, significantly reducing the time and complexity involved in analyzing Signal's encrypted data.

iii) **Comprehensive CSV/HTML Reports for Forensic Analysis:** `SignalForensics` generates detailed CSV and HTML reports that record the forensic artifacts found in Signal's local files. These reports transform raw decrypted data into a structured and accessible format, making it easier for forensic investigators to analyze and interpret the evidence.

iv) **Support for Expired or Deleted Messages:** Unlike when reading directly from the application, our approach enables the extraction of content that would otherwise no longer be available, such as expired or deleted messages. This ensures that valuable evidence is preserved even after the intended lifespan of the data within the application has passed.

v) **Forensically Sound Artifact Retrieval:** The proposed methodology and tool facilitates artifact retrieval in a manner that avoids interference with the data, maintaining the integrity of forensic evidence to the greatest extent possible. Although the process requires the operating system to be active to access encryption keys via DPAPI, it avoids directly launching Signal Desktop, preventing any modifications to user data within the application. For example, view-once media messages, which normally self-destruct upon being opened in the official application, can be extracted and preserved using this approach. This allows investigators to securely access and analyze such media without triggering their deletion, ensuring availability of otherwise ephemeral content.

vi) **Addressing Key Cryptographic Challenges:** Although `SignalForensics` requires access to the execution environment or the ability to decrypt DPAPI-protected keys externally, it still represents a significant advancement in forensic analysis by automating previously labor-intensive processes and providing new insights into Signal's encryption mechanisms and data storage practices.

The remainder of this paper is organized as follows. Section 2 reviews existing research in this field. Section 3 introduces key concepts and explains how certain Signal features are implemented, providing essential context for the following sections. Section 4 outlines the hardware, operating systems, and software tools used in this study, as well as the methodology followed to achieve the research objectives. Section 5 details the locations of relevant files within Signal Desktop and describes the decryption scheme, employed by the application, that must be mimicked to decrypt these files. Section 6 examines the forensic artifacts retrievable from Signal Desktop's local files and explains their interrelations, detailing the forensically relevant information that can be extracted from them. Section 7 describes the developed script, which automates Signal Desktop's decryption process and generates informative reports. Finally, Section 8 concludes the paper. Table 1 provides the list of acronyms used throughout the paper.

2. Related work

Given that Electron applications run on the Chromium engine (OpenJS Foundation and Electron contributors, 2023), it is relevant to review the existing literature on the cryptographic methods implemented in this platform, as well as forensic analyses conducted in similar environments. This section reviews the existing literature, addressing methodologies, challenges, and key findings pertinent to the domain.

Karo Karo et al. (2024) examined the forensic artifacts generated by the Signal Desktop application on Windows 11 under three distinct scenarios: i) Sending messages, photos, and PDFs without deleting messages; ii) Deleting messages, photos, and PDFs after sending; iii) Uninstalling the Signal Desktop application. Although no chat logs were successfully recovered in any of the scenarios, all shared attachments were retrieved in the first case, whereas only some were recovered in the second. The study mainly focused on applying the IDFIF v2 (Kohn et al., 2013) methodology to a real-world scenario and utilizing artifact

[3] https://threema.ch.
[4] https://wire.com.
[5] https://briarproject.org.
[6] https://getsession.org.
[7] https://simplex.chat.
[8] https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms.
[9] https://apps.apple.com/app/signal-private-messenger/id874139669.
[10] https://signal.org/download.

acquisition tools, such as FTK Imager and Autopsy, for data collection. However, no in-depth analysis of the generated artifacts was conducted.

Bilz (2021) investigated the forensic artifacts generated by **version 5.4.1** of the Signal Desktop application on Windows 10. They observed that the directories within Signal's Roaming and Local folders follow the expected structure of an Electron-based application, something we do confirm in our work. According to Bilz (2021), the database present in the `sql` directory within Signal's Roaming folder contains most of the relevant forensic data. Although the database format is SQLCipher 4 — an open source SQLite extension that provides transparent 256-bit AES full database encryption (Zetetic, LLC, 2024) — the encryption key is stored in plaintext within a JSON configuration file (Bilz, 2021). Using the DB Browser for SQLCipher tool and this key, the investigator gained full access to the contents of the encrypted database. However, when we applied this methodology in **version 7.32.0** of the application, it yielded different results. It was impossible to decrypt the database due to changes in the storage of cryptographic keys or the way data is encrypted.

Other works involving forensic analysis of applications with instant messaging capabilities built on top of Electron include Gupta et al. (2024), which analyzes Discord, and Bowling et al. (2023), which focuses on Microsoft Teams.

Although non-Windows and non-Electron related, Son et al. forensically analyzed the Android version of three instant messaging applications, one of which was Signal, version **5.19.4** (Son et al., 2022). They exported the messenger backup migration from the non-rooted Android device to a rooted one, thus leveraging root access to study the Signal application. The authors also developed an Android application designed to interact with the Android Keystore leveraging key stored in the Keystore to perform decryption.

Kim et al. (2025) investigated the forensic artifacts generated by the Web and Universal Windows Platform (UWP) versions of WhatsApp, analyzing their encryption schemes and proposing decryption methods. Their study revealed that while the Web version stores most decryption-related elements in the browser's storage, additional steps were required to fully decrypt the local files. Similarly, for the UWP version, they identified a complex encryption mechanism in which the database encryption key was secured through multiple layers, necessitating an alternative approach to retrieve it without executing the application. Despite these challenges, the researchers eventually succeeded in decrypting the UWP version using only locally stored data. They discovered that the database encryption key was quadruple-encrypted and required a call to the UWP API for decryption. However, since running the application and hooking into it to collect data is impractical in real-world forensic scenarios, they analyzed the detailed operation of the API and proposed a method to collect the data required without invoking the API.

Taneva et al. (2023) provided an overview of the encryption process for cookies in Google Chrome on Windows systems. The researcher observed that Chrome employs a combination of AES-256-GCM[11] encryption and DPAPI[12] protection to secure the database containing the browser cookies. Although the study offers valuable insights, it only briefly touches upon this encryption process. A closer examination will be relevant to our investigation since both Chrome and the Electron framework — in which Signal Desktop is based on — are built on the Chromium engine (OpenJS Foundation and Electron contributors, 2023; The Chromium Project, 2025).

In July 2024, Signal addressed a significant problem regarding the decryption key for the Desktop applications' local database. Previously,

```
{
  "key": "7eeec7c5f94553a2e6caf6665e4255c4c427ca2be6d6f
   ↪  edfb25ba0bc803fd231",
  "mediaPermissions": true,
  "mediaCameraPermissions": true
}
```

Fig. 1. An example of the contents the `config.json` file had when the SQLCipher key was stored unprotected.

```
{
  "id": "0c66f702-102c-45f2-b8f6-367b57da7970",
  "groupId":
   ↪  "5AxQsdshh8XupFF8gwQPFY6QIAWz6c+AyTxugrBl0cQ=",
  "type": "group",
  "name": "Test Group"
}
```

Fig. 2. Excerpt of a group conversation's information stored in the `json` column of the `conversations` table.

this key was stored in plain-text on the local disk, which presented a potential vulnerability if an attacker gained access to the file system (Abrams, 2024). Fig. 1 exemplifies the contents of a `config.json` file generated by an old Signal Desktop version. An X (formerly Twitter) user,[13] demonstrated that it was possible to usurp the session of a Signal user by copying the application's data from one system to another (Abrams, 2024). Following public scrutiny, Signal implemented a fix by adopting Electron's `safeStorage` API. This update introduces an opportunity for forensic and cybersecurity analysis, warranting a re-evaluation of Signal Desktop's resilience against potential attack vectors and its suitability for forensic artifact collection.

3. Background

Signal is a centralized end-to-end encrypted instant messaging application. As outlined in the Introduction section, Signal incorporates various security and privacy-oriented solutions. Most of these operate seamlessly in the background, remaining largely unnoticed by the user. In this paper, the focus is on the features that users actively perceive and interact with, which in turn generate valuable forensic artifacts. This section explores the implementation of some of these features within Signal, explaining key concepts that will be referenced throughout the paper.

Conversations. In Signal's context, a "conversation" refers to a *chat room* – a place where messages can be exchanged. A chat room can be either a one-on-one private channel between two users or a group chat where up to 1000 users can communicate with each other (Signal Foundation, 2025a). These can be distinguished by the type of conversation, which will be "private" for the former and "group" for the latter, as well as the differing metadata each contains. For example, as evidenced in Figs. 2 and 3, only a group conversation will have a "groupId" field, whereas only a private conversation will feature a "serviceId" field. It is relevant to note that the "Notes to self" function is handled under the hood as a single-user private chat that is only accessible by the current user.

Contacts. In Signal, the concept of a "contact" does not exist as a distinct internal entity. While the user interface provides the impression of a contact list (*e.g.*, when adding group members, or when sending the first message to a recipient), Signal manages contacts as private chat conversation instances. As described in the previous subsection, they can be identified by their `serviceId` field. These conversations are

[11] Galois/Counter Mode is a mode of operation for symmetric-key cryptographic block ciphers, designated authenticated encryption, as the name suggests it provides both encryption and authentication by combining counter mode encryption with Galois field multiplication.

[12] DPAPI, or Data Protection API, is a Windows service for encrypting sensitive data, using user or system keys for protection.

[13] https://x.com/mysk_co/status/1809287118235070662.

```
{
  "id": "bbf29929-62ac-4371-aa2b-ef8f50fe168d",
  "serviceId": "1ed50204-ea7f-498b-8bfa-6999f63c4143",
  "e164": "+35191XXXXXXX"
  "type": "private",
  "name": "Person A"
}
```


Fig. 3. Excerpt of a private conversation's information stored in the `json` column of the `conversations` table (phone number redacted for privacy).

what we refer to as "contacts" in this paper. When the user has not yet received any messages from a particular Signal contact, the `serviceId` corresponds to that contact's phone number identifier (PNI). However, once the user receives a message, the `serviceId` changes to an account identifier (ACI) — *e.g.*, `1ed50204-ea7f-498b-8bfa-6999f63c4143`.

Stories. Similar to other social media and instant messaging applications, a "story" is a temporary post consisting of text, images, or videos, shared with selected contacts. These posts are only visible for a limited duration of 24 hours,[14] after which they automatically disappear. In Signal, an outgoing story is handled as a single message sent to the selected contacts mentioned above. Likewise, an incoming story is regarded as a message received from the contact who shared the story. In both scenarios, the message instance will have the `type` field set to `story`.

View-once media. This feature enables users to send messages containing individual photos or videos that are automatically removed from the conversation after being viewed. Although sending these messages is exclusive to the Android and iOS apps, the Desktop version can still receive and view these messages. These messages retain their standard `incoming` or `outgoing` type, depending on whether they were received or sent by the current user, but they are distinguished by the `isViewOnce` field being set to `true`.

Current user. Throughout this article, the term "current user" is frequently used. This refers to the user whose account was logged into the Signal Desktop application at the time the analyzed artifacts were generated. It distinguishes the "current user" from the broader term "user" which, unless otherwise specified, refers to any Signal user.

4. Materials and methods

This section describes the hardware and software environment used in this study.

4.1. Hardware

The research was conducted using systems running Windows 11. Additionally, an Android mobile device was utilized to set up the Signal account, as the Signal Desktop application requires an initial connection to a mobile device for account creation. This requirement arises from Signal's design, which mandates a phone number for account registration. While the mobile device is not necessary for receiving messages on Signal Desktop once the account is established, it remains essential for the initial registration process. Table 2 provides an overview of the operating systems used in the study, along with their respective versions.

4.2. Software

A wide range of tools played a critical role throughout the study, serving various purposes such as reverse engineering, data analysis, and environment simulation. For instance, CyberChef, with its versatile "recipe" system for building custom processing pipelines, was invaluable for encoding, decoding, and cryptographic operations across multiple stages of the research. NirSoft's DataProtectionDecryptor — a Windows tool that exposes DPAPI decryption capabilities — proved essential during the reverse engineering of Electron's safeStorage API on Windows. Table 3 enumerates all the tools employed along with their respective versions, while Table 4 documents the Signal Desktop versions used in our study. No relevant differences were identified between the examined Signal Desktop versions that would significantly impact our methodology or the findings of this study.

Table 2
List of operating systems used in the study and their respective versions.

OS	version	Build number
Windows 11	24H2	26100.3194
Windows 11	23H2	22621.4317

Table 3
List of the software used in this study and their respective versions.

Software	Version
DB Browser	3.13.1
NirSoft's DataProtectionDecryptor	1.13
HxD	2.5.0.0
CyberChef	10.19.4
Visual Studio Code	1.96.4

Table 4
List of Signal Desktop versions used in this study.

Version	Release Date	Version	Release Date
7.32.0	Nov 6, 2024	7.36.1	Dec 18, 2024
7.33.0	Nov 13, 2024	7.37.0	Jan 8, 2025
7.34.0	Nov 20, 2024	7.38.0	Jan 15, 2025
7.35.0	Dec 4, 2024	7.39.0	Jan 23, 2025
7.35.1	Dec 6, 2024	7.40.0	Jan 29, 2025
7.36.0	Dec 12, 2024	7.40.1	Feb 2, 2025

4.3. Method

The methodology adopted in this study is divided into two distinct phases: i) the decryption of Signal Desktop's locally stored data and ii) the analysis and documentation of digital forensic artifacts.

Each phase aimed to address different research objectives, ensuring a comprehensive understanding of the application's data storage mechanisms and the artifacts it generates. Additionally, as a secondary objective integrated into both phases, this study aimed to develop an automated Python script capable of decrypting Signal's local data and processing its forensic artifacts.

The first phase focused on decrypting the application's local data, which involved a combination of reverse engineering and cryptographic analysis. The steps undertaken were as follows:

1. **Analysis of Signal and Its Dependencies**: An in-depth examination into the application's architecture was conducted, including an analysis of its underlying frameworks — *i.e.*, Electron and Chromium. This investigation provided the necessary insights into the application's data storage and encryption mechanisms, which facilitated the mimicking of its local data decryption process.
2. **Collection of Cryptographically Relevant Data**: Key cryptographic materials, including encryption keys stored in specific configuration files, were identified and extracted. Additionally, the

[14] https://support.signal.org/hc/articles/5008009166234-Stories.

encrypted local files (*i.e.*, the cipher data) were recovered from the application's local storage in preparation for the next step.
3. **File Decryption**: Using the obtained cryptographic materials and the knowledge gained from the initial analysis, the encrypted files were successfully decrypted. This step confirmed the viability of the devised decryption process and validated the approach.
4. **Documentation of a Replicable Decryption Process**: Once the decryption process was confirmed to be effective, a detailed and structured method, described in section 5, was formulated. This plan was designed to simplify the decryption process, ensuring that it could be easily replicated by other forensic investigators.
5. **Automation of the Decryption Process**: To streamline the devised decryption process, a Python script — described in section 7 — was developed based on the previously established method. The script `SignalForensics` automates the extraction of cryptographic materials, decryption of files, and reconstruction of the original data. `SignalForensics` was designed to replicate the decryption process from various possible starting points, with minimal manual intervention, ensuring versatility and efficiency in future use.

The second phase involved the examination of the decrypted files and the identification of digital forensic artifacts generated by the application. The steps included:

1. **Analysis of Decrypted Files**: The decrypted files were analyzed to identify the types of information they contained. Particular attention was given to data that could be related to the user or their actions, such as activity records, message contents, and system logs.
2. **Simulation of User Activity**: To generate a comprehensive set of forensic artifacts, various tests were conducted on the application. These tests simulated real-world user interactions, such as sending messages, making calls, and posting stories. The goal was to produce a diverse variety of data, including logs, caches, user metrics, and temporary files, representative of what one might encounter in a real forensic investigation.
3. **Reanalysis and Comparison of Artifacts**: The artifacts generated during the simulated user activity were compared with those identified in the initial analysis. This comparison aimed to establish a correlation between specific application functionalities and the artifacts they produce. For example, arbitrary values in certain fields (*e.g.*, 1, 2 or 3) were mapped to their corresponding meanings or actions within Signal.
4. **Documentation of Artifacts**: All identified artifacts were documented, detailing their locations, functionalities, and the relevant information they contained. This documentation provides a detailed reference for forensic investigators, enabling them to interpret and utilize these artifacts effectively in real-world scenarios.
5. **Automated Reporting of Forensic Artifacts**: Building upon the written documentation of the previous step, a reporting module was added to `SignalForensics`. This functionality generates detailed reports that summarize the identified artifacts in CSV and HTML format. The reporting module was designed to assist forensic investigators by extracting raw data and key insights from the application's forensic artifacts and presenting them in a structured, accessible format, thereby enhancing the efficiency and clarity of forensic analysis.

5. Decrypting signal artifacts

Signal Desktop stores various files locally to ensure smooth operation, data synchronization, and some offline functionality. Forensic analysis of these files requires identifying their locations and understanding how they are secured. This section outlines the structure of the relevant files and explains the decryption process used to access them.

5.1. Important file locations

Before delineating how Signal secures its local files in the desktop application, it is important to understand which files exist and where they are located.

On Windows, all the relevant files are located in the users' personal Roaming folder — whose location is stored in the `%APPDATA%` environment variable. Henceforth, we refer to this folder as the "data directory". The data directory is structured in the following manner:

```
%APPDATA%
└─ Signal/
   ├─ attachments.noindex/
   │  ├─ 00/
   │  │  └─ ...
   │  ├─ 01/
   │  │  └─ ...
   │  └─ ...
   ├─ avatars.noindex/
   │  └─ ...
   ├─ drafts.noindex/
   │  └─ ...
   ├─ sql/
   │  └─ db.sqlite
   ├─ config.json
   └─ Local State
```

The `attachments.noindex` directory stores exchanged attachments and the current avatar of the user, contacts, and groups. The directory functions as a hash table in folder format, containing up to 256 subdirectories named after each possible hexadecimal value of a byte (*i.e.*, 00 to FF). Each file is assigned a filename based on the hexadecimal representation of 32 bytes of random data and is placed in the subdirectory corresponding to the first byte of the filename (*i.e.*, its key in the hash table). The directories `avatars.noindex` and `drafts.noindex` follow the same structure, with the former storing user-uploaded avatars — *i.e.*, avatars the user has uploaded to Signal, including those no longer in use — and the latter containing attachments included in the draft messages

The `sql` directory contains a SQLCipher database named `db.sqlite`. This encrypted database stores various valuable forensic artifacts, including user's personal information, metadata, exchanged messages, and the keys and nonces required to decrypt the attachments and avatars.

Finally, the `config.json` and `Local State` files, located in the `Signal` directory, hold the cryptographic material required to decrypt the SQLCipher database. The `config.json` file contains the SQLCipher key in encrypted form, while the `Local State` file stores a DPAPI-protected key that is used to decrypt it. For easier comprehension, we refer to the latter as the "auxiliary key" in the upcoming sections.

5.2. Decryption process

To examine the artifacts generated by Signal, it is first necessary to decrypt them. This section outlines the decryption process employed by Signal, providing a methodology for manually replicating it. We split this process in two steps. Both are illustrated in the flowcharts presented in Fig. 4 and Fig. 5. The first step covers the decryption of the SQLCipher key and its use to retrieve data from the SQLCipher database. The second step demonstrates how the cryptographic materials stored in the database can be used to decrypt exchanged attachments, draft message attachments, and avatars.

5.2.1. First step: decrypting SQLCipher database

As referenced in Related Work, Signal adopted Electron's `safeStorage` API to safely handle the SQLCipher key. Our analysis of

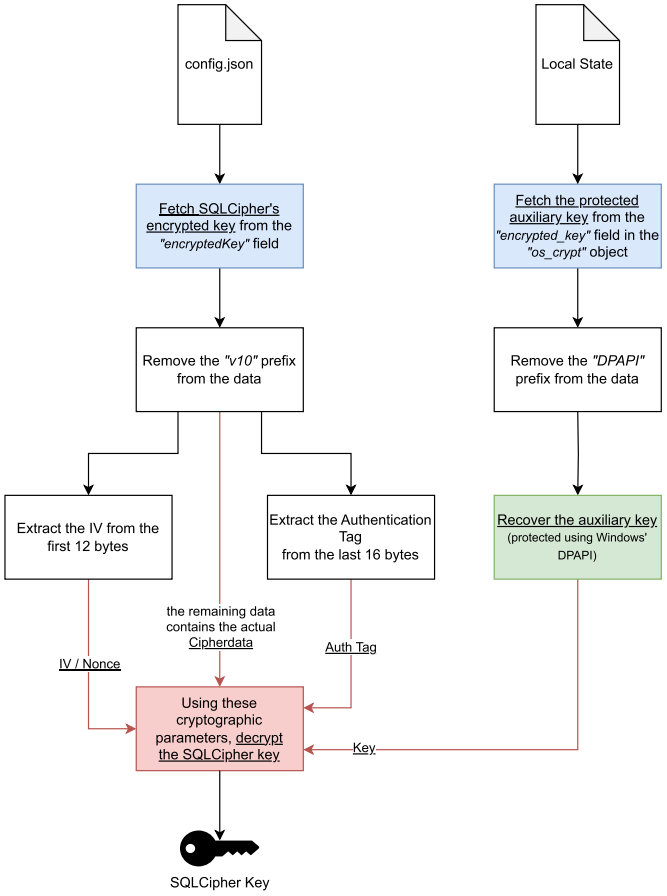

Fig. 4. Flowchart illustration of the decryption process for the SQLCipher database.

the source code, found in Signal Desktop's GitHub repository,[15] confirms that no further manipulation of the plain or cipher data is performed beyond this. Given this, we believe the following methodology is applicable to any Electron-based application utilizing the safeStorage API.

Signal retrieves the cipher data from the `encryptedKey` field in `config.json`, and then calls Electron safeStorage's `decryptString` function,[16] passing the cipher data as a parameter. The `decryptString` function[17] validates the cipher data before invoking Chromium's `OSCrypt::DecryptString`. Chromium's OSCrypt implements encryption and decryption functions that leverage platform-specific cryptographic services such as the Data Protection API (DPAPI) on Windows and Keychain on macOS. As stated earlier, in this paper, we focus on the Windows environment.

The `DecryptString` function[18] starts by retrieving the base64-encoded auxiliary key cipher data from the `os_crypt.encrypted_key` field in `LocalState`.[19] Fig. 6 exemplifies the content stored in this field after decoding it using base64.

[15] https://github.com/signalapp/Signal-Desktop.
[16] https://github.com/signalapp/Signal-Desktop/blob/ac09a508f1085c58afdeb6c758ccb22ab8ed8e51/app/main.ts#L1690C4-L1690C48.
[17] https://github.com/electron/electron/blob/6e3a5daf62314d4de541d64a1098b0bb3a37168c/shell/browser/api/electron_api_safe_storage.cc#L118.
[18] https://github.com/chromium/chromium/blob/7d17d557f6f637cbb4520f30297b1fde9e3cfc54/components/os_crypt/sync/os_crypt_win.cc#L216.
[19] https://github.com/chromium/chromium/blob/7d17d557f6f637cbb4520f30297b1fde9e3cfc54/components/os_crypt/sync/os_crypt_win.cc#L111.

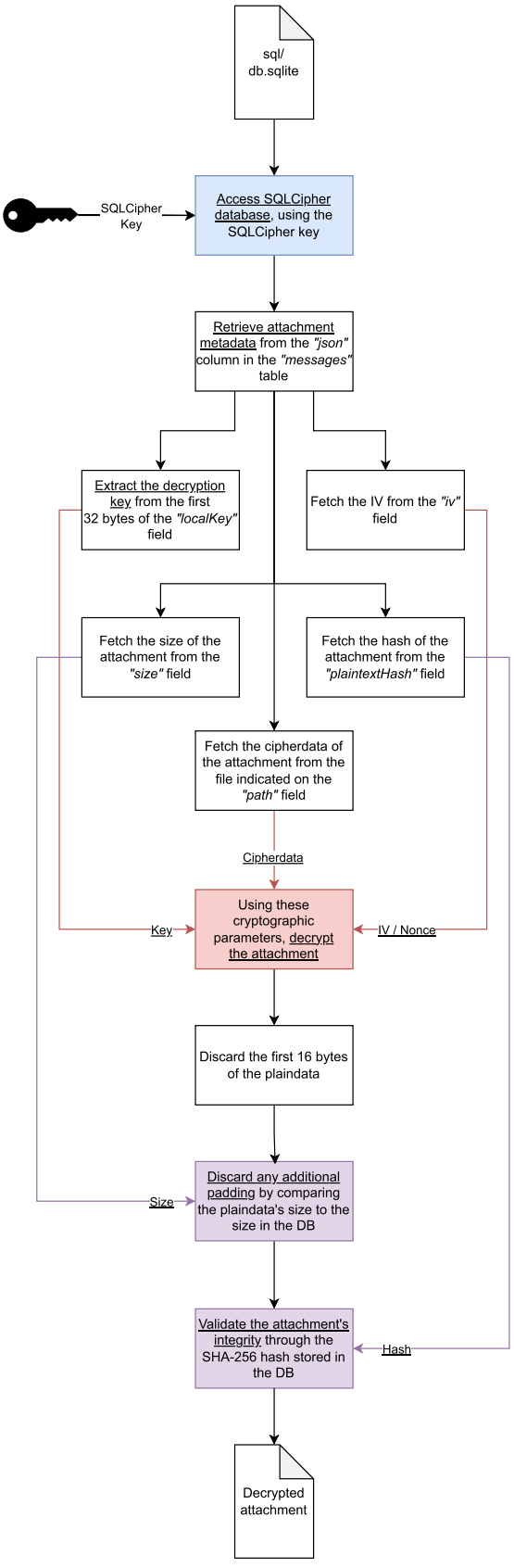

Fig. 5. Flowchart illustration of the decryption process for attachments and avatars.

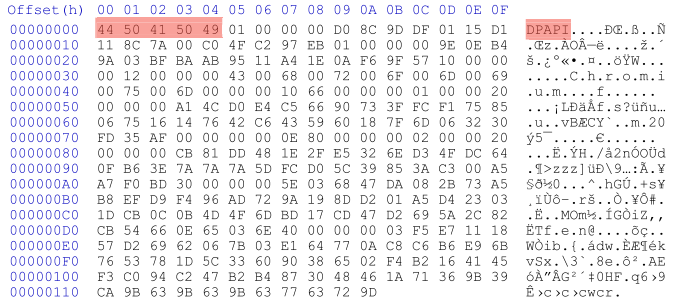

Fig. 6. The base64-decoded encrypted auxiliary key stored in the `Local State` file.

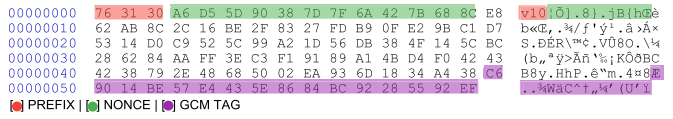

Fig. 7. The encrypted SQLCipher key stored in the `config.json` file.

After discarding the "DPAPI" prefix, the auxiliary key is decrypted through the DPAPI[20] and kept in volatile memory for future use. To replicate this step, we utilized NirSoft's DataProtectionDecryptor[21] tool. With the auxiliary key, the `DecryptString` function decrypts the SQLCipher key using the AES-256 algorithm in Galois/Counter Mode (GCM)[18] For this, it still requires two extra parameters, the nonce and the authentication tag. To acquire them, we must examine the data acquired from the `config.json` file. As we can ascertain in the source code and observe in Fig. 7, the first three bytes are the `v10` prefix, the nonce is located in the following 12 bytes, the actual cipher data occupies the rest of the data up until the last 16 bytes where the authentication tag is stored.

Using the decrypted SQLCipher key, it is now possible to access and query Signal's local database through tools like DB Browser for SQLCipher.

5.2.2. *Second step: decrypting attachments and avatars*

While Signal handles the decryption of attachments and avatars separately, their decryption processes are nearly identical. For the sake of simplicity, we will first describe the more complex attachment decryption process and then highlight the key differences with avatar decryption. Note that internally, view-once messages have their media handled as a normal attachment, following the exact same procedure for the decryption of their contents.

The table `messages`, as the name suggests, stores information on the messages the user exchanged. Messages containing attachments have the `hasAttachments` field set to 1. The `json` field holds the message data in JSON format. This field will be our focus moving forward. Within the JSON key `attachments`, we find an array that stores details about each attachment included in the message. As illustrated in Fig. 8, the `path` field provides the path to the encrypted attachment file, while the `localKey` and `iv` fields contain the base64-encoded decryption key and nonce (initialization vector), respectfully. Also relevant are the `size` and `plaintextHash` fields — the `size` field specifies the attachment's size in bytes, while the `plaintextHash` contains the SHA-256 hash of the decrypted attachment.


```
{
  "size": 543394,
  "contentType": "audio/mpeg",
  "localKey": "pDFricod8L7qAzVJuAMCHfVHyFCF/kDGyfGBYPMW
   ↪ Kg6SZpfAIw/4OiqlNSWtvVTZCcFcN3TyJGKSO8UUck4e4w==",
  "iv": "Z78XQDc6YwjPpLbyVzSz0w==",
  "path": "c1\\c24c12ebd76a783a547d74ddbf8d81376823bc21
   ↪ fe10d4183f3c87841d744841",
  "plaintextHash": "0be6c47276804a39e30dd2674f8b26eb3f5
   ↪ 1bab3efee4f1bf715e2fa18b15e34",
}
```


Fig. 8. An example excerpt of attachment information stored in the `json` column of the `messages` table.

Signal encrypts[22] attachments using the AES-256 algorithm in the Cipher Block Chaining (CBC) mode. The data extracted from `localKey` is 64 bytes long; the AES-256 decryption key is located in the first 32 bytes. After decrypting the attachment's cipher data, the first 16 bytes of the decrypted data must be discarded, along with any additional padding. Trailing bytes that surpass the attachment size, as indicated by the `size` field, are to be discarded to establish the precise length of padding requiring removal. Finally, to guarantee the integrity of the attachment, the SHA-256 hash of the remaining data is compared with the value in the `plaintextHash` field.

For draft attachments and avatars, the decryption process is largely similar to that of exchanged attachments, having two key differences. The first difference is that their cryptographic metadata is stored in the `json` field of the `conversations` table, specifically in the record corresponding to the conversation to which they pertain.

- The `draftAttachments` key contains an array with the cryptographic details of all draft attachments.
- The `avatar` key stores the cryptographic metadata of the current avatar for the conversation.
- The `avatars` key is populated only in group chats and the "Notes to self" conversation. This key maintains a list of cryptographic information for all avatars that the user has uploaded for the group or themselves.

The second difference is that none of these entries include the `iv` field. Instead, a 16-byte array of `0x00` bytes should be used as the initialization vector during decryption.

6. Forensic artifacts

In this section, we detail the main forensic artifacts of Signal Desktop. To populate the application's database with realistic content, we simulated user activity by exchanging messages between two devices: a Windows 11 desktop running Signal Desktop and an Android 13 smartphone running Signal. Both applications were updated to the most recent available version at the time of testing. The exchanged messages included plain text (including text formatting and mentions), images, audio recordings (*i.e.*, voice messages and MP3s), videos, and other files. Additionally, features such as group creation, message deletion, story posting, and calls were tested to ensure the generation of diverse and representative forensic artifacts.

Most of the valuable forensic artifacts identified during our research were located within Signal Desktop's SQLite database. As delineated in Section 5, this database is encrypted at rest using SQLCipher and resides in the folder created by Signal within the Windows user's Roaming directory, specifically at the relative path `sql/db.sqlite`.

[20] https://github.com/chromium/chromium/blob/7d17d557f6f637cbb4520f30297b1fde9e3cfc54/components/os_crypt/sync/os_crypt_win.cc#L294C8-L294C30.

[21] https://www.nirsoft.net/utils/dpapi_data_decryptor.html.

[22] https://github.com/signalapp/Signal-Desktop/blob/94dba11bcb3973b98784b0a7aaefa6ea03c330ea/ts/AttachmentCrypto.ts#L380.

Table 5
Brief description of relevant tables in the SQLCipher database.

Name	Nr. of Columns	Description
`callsHistory`	10	Stores information on the call records of the current user.
`conversations`	14	Stores information on the conversations of the current user.
`items`	2	Stores information on the current user and the application's settings.
`mentions`	4	Stores information on mentions present in the current user's exchanged messages.
`messages`	44	Stores information on the current user's exchanged messages.
`reactions`	9	Stores information on the emoji reactions to the current user's exchanged messages.

The database contains 45 tables, 76 indices and 6 triggers. In this section, we focus on the tables and associated artifacts that are forensically relevant: `callsHistory`, `conversations`, `items`, `mentions`, `messages` and `reactions`. Table 5 gives a brief description of each of these tables. Unless otherwise stated, all timestamps mentioned in this section are in Unix epoch time milliseconds and universal time coordinated (UTC).

6.1. Items table

The `items` table stores a wide range of information, including user settings such as enabling or disabling specific features, metadata related to blocked groups and users, as well as information on the current user – *e.g.*, ACI and phone number (in E.164 format). The `id` column stores the identifier of the attribute recorded in the `json` column, which can represent user settings, metadata, or account-related information. The `json` column contains a JSON object with two fields: `id` and `value`. The `id` field mirrors the value in the `id` column, while the `value` field holds the corresponding data. This data can take various forms, including strings, lists, base64-encoded content, objects, etc.

During our research and tests, we identified 84 distinct records in the `items` table. Table 6 shows the most relevant entries from a forensic perspective — *i.e.*, attributes that reflect user configuration, identity, or interactions with other users.

6.2. Conversations table

The `conversations` table stores information about the user's conversations. The `json` column stores the conversation's metadata as a JSON string. The remaining columns serve auxiliary roles, repeating key details from the JSON object, such as the conversation's name and type. This design likely facilitates the execution of SQL operations on conversation data without the need to parse the JSON.

After parsing the JSON object, we encounter various fields. In this section, we will only note the most interesting ones. Some of these fields are present in private conversations, others in group conversations, but most are present in both. There is still a possibility that the presence of a few of these fields might vary on a stricter case-by-case basis (*e.g.*, a draft message existing or not); for that reason, caution is recommended when parsing these objects.

Table 7 details the fields that are expected in every conversation. Table 8 details the fields that are expected to be present in private conversations. Finally, Table 9 details the fields expected in group conversations. Be sure to check footnotes [23] and [24] for additional context in some of the fields' descriptions.

Table 6
List of relevant attributes found in the database's items table.

Attribute	Type	Contents
`blocked`	List	The phone numbers (E.164) of users blocked by the current user.
`blocked-groups`	List	The IDs of groups blocked by the current user.
`blocked-uuids`	List	The ACIs of users blocked by the current user.
`device_name`	String	The name of the device Signal Desktop is installed on.
`hasStoriesDisabled`	Boolean	True if the user disabled the stories functionality.
`number_id`	String	The phone number (E.164) of the current user.
`pni`	String	The PNI of the current user.
`read-receipt-setting`	Boolean	True if the current user has read receipts enabled.
`typingIndicators`	Boolean	True if the current user has typing indicators enabled.
`unreadCount`	Integer	Total number of unread messages throughout all conversations.
`uuid_id`	String	The ACI of the current user.
`version`	String	The version of Signal Desktop that last accessed this database.

Besides `profileAvatar` and `avatar`, there is a third field where information on older avatars can be found: the `avatars` field. This field is only present in group conversations and in the "Notes to self" private conversation. It contains a list of objects that, along with simple icons,[25] stores the cryptographic parameters of avatars the current user has uploaded for the group (if on a group conversation) or previous user avatars (if on the "Notes to self" conversation). These are the avatars that pop-up as options when the current user attempts to change their or a group's avatar. The avatars' decryption process is detailed in section 5.2.2.

6.3. Calls history table

The `calls_history` table logs the user's call history.

This table includes four key ID columns: `callId`, `peerId`, `ringerId`, and `startedById`. The first serves as the table's primary key. The data stored in the latter three tables varies based on the call mode and direction. Signal supports two call modes — "Direct" and "Group" — which are identified by the `mode` column. In direct calls, which occur between two users, the `peerId` represents the other participant. In group calls, which are initiated within a group chat, the `peerId` corresponds to the group's unique `groupId`. The direction of the call, recorded in the `direction` column, indicates whether it was outgoing — initiated by the user — or incoming — received by the user. In incoming calls, the `startedById` stores the account identifier (ACI) of the user who initiated the call, while the `ringerId` contains the ACI of the account that last rang the user – in incoming direct calls this column is redundant. In outgoing group calls, both the `startedById` and `ringerId`

[23] This includes messages that have expired or were deleted.
[24] Given the aforementioned fields interact and complement each other similarly to how they do in a standard message, detailed explanations will be deferred to the messages section later in this document.
[25] Icons that are made up of a symbol and a foreground color.

Table 7
Conversation fields that are expected in every conversation.

Name	Content
`id`	The UUID of the conversation.
`type`	The type of conversation.
`name`	The name of the conversation (either the contact's name or the group's name).
`unreadCount`	The number of unread messages.
`messageCount`	The total number[23] of messages that were exchanged.
`sentMessageCount`	The total number[23] of messages sent by the current user.
`active_at`	The timestamp for the last activity.
`lastMessageReceivedAtMs`	The timestamp of the last exchanged message.
`lastMessageAuthor`	The name of the author of the last message. When the last message was sent by the current user, this field will have a localized version of the word "You".
`lastMessage` `lastMessageBodyRanges`	The contents[24] of the last message.
`lastMessageDeleted-ForEveryone`	A flag that describes whether the last message was deleted for everyone — *i.e.*, is a "this message was deleted" notice.
`lastMessagePrefix`	An emoji representing the content of the last message (*e.g.*, a microphone emoji for voice messages). This field is not present on text messages.
`draftTimestamp`	The timestamp of the draft message, when one exists.
`draft` `draftBodyRanges` `draftAttachments`	The contents[24] of the draft message.
`expireTimer`	The current expire timer for this conversation. When an expire timer is defined, new messages exchanged in a conversation expire (and are deleted) after the number of seconds in this field.
`isArchived`	A flag that describes whether the conversation is archived.

columns store the ACI of the current user; however, for outgoing direct calls, both columns are left empty.

The `status` column records the current status of a call. An example is the `Deleted` status, which indicates that the call notification message was removed or expired from the chat. Certain call statuses have specific meanings depending on the call mode and direction. For instance, a group call with an outgoing direction and a `Missed` status implies that none of the group members accepted the call. However, an incoming group call with the same status signifies only that the current user did not join, even though other members might have. Additionally, some statuses are exclusive to specific call modes. Only direct calls can have a `Pending` status, whereas only group calls may include statuses such as `Ringing`, `OutgoingRing`, `Joined`, and `GenericGroupCall`.

The remaining two columns — `timestamp` and `endedTimestamp` — store chronological data. The `timestamp` column records the timestamp of when the call was placed. The `endedTimestamp` column does not appear to be implemented, as it was consistently empty in all the tests we conducted.

Table 8
Conversation fields that are expected in private conversations.

Name	Content
`serviceId`	The Service ID of the contact which, as described in section 3, is either the contact's PNI or ACI.
`profileName` and `profileFamilyName`	First and last name of a contact. This is the name the contact has set for their Signal account.
`systemGivenName` and `systemFamilyName`	First and last name of a contact. This is the contact name found in the current user's external contact book.
`nicknameGivenName` and `nicknameFamilyName`	The nickname the current user gave to this contact.
`note`	A note the current user wrote in the profile of this specific contact. A note is only visible to the user who wrote it.
`e164`	The phone number of the contact in E.164 format.
`username`	The username of the contact in Signal.
`profileAvatar`	The cryptographic parameters for the contact's avatar.

Table 9
Conversation fields that are expected in group conversations.

Name	Content
`groupId`	The ID of the group. This ID is 32 bytes of random data represented as a base64-encoded string.
`description`	The description of the group.
`membersV2`	A list containing information about the group's members. Each member is saved as an object that contains their ACI in the `aci` field, and their role as an integer in the `role` field. Currently, there are only two roles in Signal, `1` for "Member" and `2` for "Administrator".
`bannedMembersV2`	A list containing information about former group members that were kicked by an Administrator. Each member object contains a `serviceId` field with their Service ID.
`avatar`	The cryptographic parameters for the groups's avatar.

6.4. Messages table

The `messages` table records information about exchanged messages, including metadata, content, cryptographic metadata of attachments, read and view states, reactions, and edit history. The `messages` table contains multiple columns; however, similarly to the `conversations` table, all the information is stored as a JSON string in the `json` field. Nonetheless, auxiliary flags such as `hasAttachments` and `hasFileAttachments` are available, making it easier to filter messages that include attachments without having to parse the JSON object.

To begin, we examine some key elements within the JSON object that are either universal across all message types, inherently self-explanatory, or straightforward to explain: Table 10. The presence of an asterisk (`*`) denotes that a field is present in all messages.

Besides the mentioned fields, a message contains several other timestamps, such as `serverTimestamp`, `decrypted_at`, and `received_at`. We chose `received_at_ms` as the standard timestamp because it proved to be the most consistent across different message types and functionalities.

Before delving further into the contents of the JSON object, it is essential to understand how certain data is structured and interrelated. For instance, message text content is stored in the `body` field, but additional details like mentions and text formatting (*e.g.*, bold and italics)

Table 10
Key elements in a message's JSON object (fields marked with an * denote they are present in all messages).

Name	Content
`id`*	The ID of the message.
`type`*	The type of message.
`conversationId`*	The UUID of the conversation the message belongs to.
`sourceServiceId`*	The Service ID of the message's author.
`sourceDevice`	An integer — *e.g.*, 1, 2, 3, etc. — representing from which of the author's known devices the message was sent.
`sent_at`*	The timestamp representing when the message was sent.
`received_at_ms`*	The timestamp representing when the message was received on the current device.
`expiresTimer`	The lifespan of the message in seconds, indicating how long it will remain before expiring. If this field is `0` or missing, the message will not expire.
`expirationStart-Timestamp`	The timestamp marking the start of the expiration countdown for this message.
`isErased`	A boolean flag denoting whether the message was deleted for everyone — *i.e.*, is a "this message was deleted" notice.
`body`, `bodyRanges`, `preview` and `attachments`	The contents of the message.

are found in the `bodyRanges` field, requiring both fields to be combined for a complete interpretation. The following subsections outline these relationships and their significance.

6.4.1. Message types

The message type can be inferred from the contents of the `type` column or `type` field in the JSON object. There are 21 possible message types Signal Foundation (2025b). However, during our tests, we only observed seven of these types. Of these, five are noteworthy, specifically:

- `incoming` – An incoming message, sent by another user and received by the current user.
- `outgoing` – An outgoing message, sent by the current user.
- `group-v2-change` – A change in a group's settings or metadata. This message type will be explained in a latter subsection.
- `timer-notification` – A change in a conversation's expire timer[26] state or duration. Messages with this type, will have a `expirationTimerUpdate` field that contains an object with two fields, the new duration for the expire timer — `expireTimer` — and the Service Id of the user who made that change — `sourceServiceId`. If the expire timer was disabled, the duration will be set to 0.
- `call-history` – The in-chat message that appears when a call is attempted. By cross-referencing the `callId` field in the message's JSON object with the `callId` column in the `calls_history` table, details about the call can be obtained.

Unless otherwise stated, the following subsections apply only to messages of type `incoming` and `outgoing`.

[26] The timer that determines when a message gets automatically deleted.

```
"bodyRanges": [
  { "length": 10, "start": 13, "style": 1 },
  { "length": 1, "start": 30, "style": 1 },
  { "length": 8, "start": 23, "style": 2 },
  {
    "length": 1,
    "mentionAci":
     ↪  "ddd8e8a2-b058-4830-a38a-98cb13e97b91",
    "replacementText": "Tiago Santos",
    "start": 39
  },
  { "length": 6, "start": 34, "style": 1 },
  { "length": 6, "start": 34, "style": 2 }
],
"body": "Test message with bold italics or both *"
```

Fig. 9. An example of the contents found in the `bodyRanges` field.

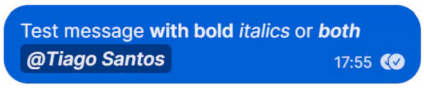

Fig. 10. An example message with text formatting and mentions displayed on the Signal Desktop client.

6.4.2. Body, mentions and text formatting

As previously mentioned, to account for user mentions and text formatting, we must parse the `body` and `bodyRanges` fields. The `body` field contains the simple text content of the message. The `bodyRanges` field is a list that stores mention details and text formatting details as objects. Fig. 9 exemplifies a `bodyRanges` list with both objects. The asterisk represents the object replacement character that would be present at the mention's location.

Mention objects consist of four fields:

1. `mentionAci` – The ACI of the mentioned user.
2. `replacementText` – The text that identifies the mentioned user (usually the contact's conversation name). When the mentioned user is the current user, this field is empty.
3. `start` – The index position of the object replacement character signaling a mention in the message's text body.
4. `length` – The length that the special character occupies.

To correctly introduce a mention in a message's body, one would have to replace the special character with the contents of the `replacementText` field. Text formatting objects consist of three fields:

1. `start` – The index position of where the format starts.
2. `length` – The length of the format.
3. `style` – An integer that identifies the style that will be applied to the matching text. `1` is bold, `2` is italic, `3` is a spoiler box, `4` is strike-through, and `5` is monospace.

The same matching text can have multiple styles applied to it. Fig. 10 illustrates the example text message from Fig. 9.

Additionally, a message object contains a `mentionsMe` boolean field, which indicates whether the message mentions the current user.

6.4.3. Preview

The `preview` field in Signal messages is used to store data for the preview that is automatically generated when a URL is included in the message body. This field is composed of a `url` field, which specifies the URL that generated this preview, a `title` field that often represents the name or heading of the linked webpage, and a `description`, which provides a brief summary of the content. Additionally, the `preview` field includes an `image` object, which stores the cryptographic metadata required to decrypt the preview's image. This image can be decrypted using the same process as regular attachments, and it is intended to

```
{
  "received_at_ms": 1733871561461,
  "editHistory": [{
      "attachments": [],
      "body": "Message Version 2",
      "bodyRanges": [],
      "preview": [],
      "received_at_ms": 1733871689441,
    },{
      "attachments": [],
      "body": "Message Version 1",
      "bodyRanges": [],
      "preview": [],
      "received_at_ms": 1733871561461,
    }],
  "editMessageReceivedAtMs": 1733871689441
}
```


Fig. 11. An excerpt of the fields found in a message with an edit history.

visually represent the URL, such as a website logo, a thumbnail, or any related content image.

6.4.4. Attachments

As mentioned in section 5.2.2, the `attachments` field contains a list that provides details about each attachment included in the message. Each object within this list includes metadata about an attachment and the cryptographic parameters necessary for its decryption.

6.4.5. Edit history

The body of a message can be edited multiple times. The `editMessageReceivedAtMs` field registers the timestamp of when the latest version of the message was received, while the `editHistory` field maintains a list of all previously received versions. Each entry in the `editHistory` list includes the message content stored in the `body` and `bodyRanges` fields, alongside the timestamp of when that particular edit was received, stored in the `received_at_ms` field. Although the `attachments` field appears in these entries, it remains empty, as editing message attachments is not currently supported. Changes to the `preview` field, which reflects updated links, may occur across different versions of a message. If a specific message has never been edited, both the `editMessageReceivedAtMs` field and the `editHistory` list will be missing from the main JSON object. An excerpt of a message with an edit history can be observed in Fig. 11.

6.4.6. Reactions

In Signal, users can react to messages using emojis; these are limited to one emoji per user and per message, however, the chosen emoji can be changed or removed at any time. The `reactions` field is a list containing objects that detail the reactions associated with a message. An example can be observed in Fig. 12. Reaction objects have four fields:

1. `emoji` – A string containing the reaction's emoji.
2. `fromId` – The Service ID of the reactor.[27]
3. `targetTimestamp` – The timestamp of the message to which the reaction applies. If the message has an edit history, this timestamp can be compared with the timestamps of all message versions to determine which specific version received the reaction.
4. `timestamp` – The timestamp of when the reaction was performed.

6.4.7. Read receipts and send status

It is important to clarify the difference between a "read" and a "viewed" message in the context of Signal. Both of these states only

[27] The user who performed said reaction.


```
{
  "reactions": [
    {
      "emoji": "[EMOJI]",
      "fromId": "dd52badb-b058-4830-a38a-98cb1352badb",
      "targetTimestamp": 1733871688069,
      "timestamp": 1733871766605
    }
  ]
}
```


Fig. 12. An example of the contents found in the `reactions` field.

Table 11
List of possible message read receipt states for messages received by the current user.

State	readStatus	seenStatus
Unread	1	1
Read	0	2
Viewed	2	2


```
{
  "type": "incoming",
  "readStatus": 0,
  "seenStatus": 2,
  "body": "A message received by the current user"
}
```


Fig. 13. An excerpt from a JSON object of an `incoming` message with read receipts information.

make sense when relating a user to a specific message; in group conversations, these states exist for each individual member.

A "read message" is any message — *e.g.*, a regular text message — that has been seen by the recipient. Distinctively, a "viewed message" refers to one of two situations — a voice message that has been seen in chat and played, or a view-once message that has been seen in chat and opened by the recipient. For example, if user A sends two messages to user B — a text message and a voice message — and user B only opens the conversation, both messages would be considered "read". However, as soon as user B plays the voice message, that specific message transitions to the "viewed" state. For simplicity, these two scenarios will collectively be referred to as "viewable messages".

Behind the scenes, Signal handles read receipts differently for `outgoing` messages and those received by the current user. Signal also tracks the read receipts of messages triggered by user actions, such as a call history message after initiating a call or a group change; however, this tracking applies exclusively to received messages.

Read receipts for received messages. For messages received by the user, the `readStatus` and `seenStatus` fields store integers that indicate whether the message has been seen or viewed by the current user. A `readStatus` and `seenStatus` value of `1` signifies that the message was neither read nor viewed by the current user. When the `readStatus` is `0` and the `seenStatus` is `2`, it indicates that the message has been read by the current user. Lastly, a `readStatus` and `seenStatus` value of `2` denotes that the message has been both read and viewed by the current user, a state that can only be observed in viewable messages. Table 11 summarizes these fields' values and their significance. Fig. 13 displays an excerpt of the JSON object of an example text message that was received and read by the current user.

```
{
  "type": "outgoing",
  "sendStateByConversationId": {
    "7fc993c9-439e-414d-b8d1-9d131493861c": {
      "status": "Read",
      "updatedAt": 1735790339678
    },
    "ec81695a-60d8-4910-86d7-d076cfd13983": {
      "status": "Viewed",
      "updatedAt": 1735790229545
    },
    "b7c18f3c-5739-45b9-83c7-d685cc611919": {
      "status": "Delivered",
      "updatedAt": 1735790227149
    },
    "6b0b2008-89b5-49d5-ab14-eb1ac562cb06": {
      "status": "Sent",
      "updatedAt": 1735790225007
    }
  }
}
```


Fig. 14. An excerpt from a JSON object of an `outgoing` voice message's send state.

Send status for outgoing messages. For `outgoing` messages, the `sendStateByConversationId` field is key. The `sendStateByConversationId` field maps each user's conversation ID — including the current user, who sent the message, and all receiving users — to an object containing the message's send status for that specific user. This inner object contains two key elements: the `status`, which denotes the current state of the message for the user, and the `updatedAt`, a timestamp indicating when the message send status was last updated for that user. The possible values for the `status` field are `Failed`, `Pending`, `Sent`, `Delivered`, `Read`, `Viewed`, and `Skipped`. As the names suggest, `Read` means that the respective user has read the message, while `Viewed` indicates that the user has played the viewable message. The combination of both of these elements, allows us to discern, for example, when a message was read or viewed by a certain user.

Fig. 14 shows an excerpt of the JSON object for a voice message sent by the current user to a group conversation. In this example, from bottom to top, the first user sent the message; the second received it but has neither seen nor played it; the third has both seen and played it; while the fourth has seen the message but has not played it yet.

6.4.8. Quote

In Signal, messages can be sent as direct replies to a previous message in the conversation — this is known as "quoting a message". When a message is a reply, information about the quoted message can be found in the `quote` field of the message's JSON object. The `quote` field — exemplified in Fig. 15 — is an object containing several key fields:

- `messageId` – The UUID of the quoted message.
- `authorAci` – The ACI of the user who sent the quoted message.
- `referencedMessageNotFound` – A boolean that is `true` if the quoted message could not be found.

Additionally, the `quote` object may contain `bodyRanges`, `attachments`, and `text` fields, which can help reconstruct parts of the quoted message when the original message is missing from the database.

6.4.9. Stories

As explained in Section 3, stories in Signal are stored as regular messages, identifiable by their message type — `story`. Stories uploaded by the current user will share the same conversation ID as the "Notes to Self" chat, which is tied to the current user's Service ID. Additionally, stories introduces a new attachment type, identified by its `contentType` being `text/x-signal-story`; this attachment type is used for


```
{
  "quote": {
    "authorAci": "7fc993c9-439e-414d-b8d1-9d131493861c",
    "attachments": [],
    "bodyRanges": [],
    "isViewOnce": false,
    "messageId": "01942478-3a94-749e-b70a-eb1ac820ca11",
    "referencedMessageNotFound": false,
    "text": "This message was quoted."
  },
}
```


Fig. 15. An excerpt of a `quote` field in a message's JSON object.


```
{
  "contentType": "text/x-signal-story",
  "textAttachment": {
    "color": 4285041620,
    "text": "Test",
    "textStyle": 1,
    "textForegroundColor": 4294967295
  },
  "size": 4
}
```


Fig. 16. An example of a Signal story consisting of a text attachment.

stories that consist of text displayed over a colored background. An example of a text attachment is shown in Fig. 16.

6.4.10. Group changes

Messages of type `group-v2-change` represent the notification messages sent in the group's conversation when a group-related event occurs; internally, this is known as a "group change". These events include changes to the group's profile — *i.e.*, name, description and avatar — changes to the group's configuration — *e.g.*, toggling join links — changes in the group's members, or the creation of the group itself.

These messages include a `groupV2Change` field, which is an object with two keys: `from`, identifying the Service ID of the user who performed the change or triggered the event, and `details`, a field containing a list. This list stores detail objects, each representing the specifics of a group change. A single message can contain multiple details objects, each representing a separate group event. Each detail object is composed of a `type` field, which identifies the type of change/event, along with auxiliary fields that provide additional information about the event. The structure and content of these auxiliary fields vary depending on the type of change described. Table 12 and Table 13 present all the currently possible group change types with a brief description of each, the corresponding details they include, and an example JSON illustrating their structure.

6.5. Redundant tables

Even though the `json` column in the `messages` table contains all the information about a specific message, executing SQL searches on this field is neither efficient nor straightforward. For that reason, Signal's local database includes several tables that store redundant information to facilitate these operations. Among them, we found it relevant to describe the `mentions` and `reactions` tables.

Mentions table. The `mentions` table stores information about mentions across all messages. The `messageId` column identifies the message containing the mention, while the `mentionAci`, `start`, and `length` columns provide the necessary metadata to render the mention as readable text, as described in section 6.4.2.

Table 12
List of possible group changes. (1/2).

Type	Description	Details	Example
`create`	Group Created	—	`"type": "create"`
`title`	Group Name Changed	• **newTitle**: A string containing the new group name.	`"type": "title", "newTitle": "A new group name"`
`description`	Group Description Changed	• **description**: A string containing the new group description. • **removed**: A boolean flag indicating whether the group's description was removed.	`"type": "description", "description": "A new group description", "removed": false`
`avatar`	Group Avatar Changed	• **removed**: A boolean flag indicating whether the group's avatar was removed.	`"type": "avatar", "removed": true`
`group-link-add`	Group Link Enabled	• **privilege**: An integer denoting if admin approval is required for members joining through the group link. `1` if yes, `3` if not.	`"type": "group-link-add", "privilege": 1`
`group-link-remove`	Group Link Disabled	—	`"type": "group-link-remove"`
`group-link-reset`	Group Link Reset	—	`"type": "group-link-reset"`
`access-invite-link`	Group Link Admin Approval Toggled	• **newPrivilege**: An integer denoting if admin approval was enabled or disabled for members joining through the group link. `1` if admin approval was disabled, `3` if admin approval was enabled.	`"type": "access-invite-link", "newPrivilege": 3`
`access-members`	Add Members Permission Changed	• **newPrivilege**: An integer denoting who can add members to the group. `2` if everyone, `3` if only group administrators.	`"type": "access-members", "newPrivilege": 2`
`access-attributes`	Modify Group Profile Permission Changed	• **newPrivilege**: An integer denoting who can modify the group's profile. `2` if everyone, `3` if only group administrators.	`"type": "access-attributes", "newPrivilege": 2`
`announcements-only`	Announcements Only Mode Toggled	• **announcementsOnly**: A boolean flag indicating if the announcement only mode was enabled or disabled.	`"type": "announcements-only", "announcementsOnly": true`

Table 13
List of possible group changes. (2/2).

Type	Description	Details	Example
`member-add`	Member Added	• **aci**: A string containing the ACI of the added user.	`"type": "member-add", "aci": "82d105f9-0b12-4557-8acb-ab7b1cb98a5f"`
`member-remove`	Member Removed	• **aci**: A string containing the ACI of the removed user.	`"type": "member-remove", "aci": "82d105f9-0b12-4557-8acb-ab7b1cb98a5f"`
`member-privilege`	Member Role Updated	• **aci**: A string containing the ACI of the target user. • **newPrivilege**: An integer denoting the target user's new role. `2` if Administrator, `1` if a regular member.	`"type": "member-privilege", "aci": "82d105f9-0b12-4557-8acb-ab7b1cb98a5f", "newPrivilege": 2`

Reactions table. The `reactions` table stores information about reactions across all messages. The information contained within this table's columns should be processed as described in section 6.4.6.

7. The SignalForensics script

The process described in the previous section is not overly complex; however, it is not entirely straightforward either. More importantly, it is time-consuming, especially when decrypting multiple attachments, as each attachment is encrypted with unique cryptographic data. With this in mind, we developed a script that automates this process and provides various modes of execution, allowing us to accommodate different operating systems. We named the Python script `SignalForensics`. Beyond decrypting and exporting Signal artifacts, `SignalForensics` also generates CSV and HTML reports based on the gathered information, offering a structured and comprehensive overview of the decrypted data. This ensures flexibility for integration into various processing programs and adaptability for diverse analytical purposes. The upcoming subsections detail the different execution modes and available options. Fig. 17 displays the syntax for the script. Fig. 18 exemplifies the terminal output of the script. Fig. 19 illustrates an example of an HTML report generated by the script. Please note that the screenshot is cropped and does not display the full dashboard, some fields are omitted from view to save space. `SignalForensics` is open-source and the source code can be obtained at https://github.com/labcif/SignalForensics.

```
SignalForensics [-m auto] -d <signal_dir> [-o
  ↪ <output_dir>] [OPTIONS]
SignalForensics -m aux -d <signal_dir> [-o
  ↪ <output_dir>] [-kf <file> | -k <HEX>] [OPTIONS]
SignalForensics -m key -d <signal_dir> -o <output_dir>
  ↪ [-kf <file> | -k <HEX>] [OPTIONS]
```

Fig. 17. SignalForensics' syntax.

```
SignalForensics -d
  ↪  C:\Users\TheUser\AppData\Roaming\Signal -o
  ↪  this_is_a_test
------------==<[ CONFIG ]>==------------
Mode: auto
Signal Directory:
  ↪  C:\Users\TheUser\AppData\Roaming\Signal
Output Directory: this_is_a_test
----------------------------------------
[i] SQLCipher Key: 9a325c733c1d19b840e765b5bafdee957797
  ↪  eea25b7aacae746f83f0c6e44479
[i] Opening SQLCipher database
[i] Exported the unencryted database
[i] Exporting attachments...
[i] Exported 192 attachments
[i] Writing reports...
```

Fig. 18. Example terminal output for SignalForensics.

Table 14
Execution modes in SignalForensics.

Execution Mode	Requirements
Windows Automatic	Requires the path to the data directory. Must be run on the same environment as Signal. Windows-only.
Auxiliary Key Provided	Requires the path to the data directory and the decrypted auxiliary key. Can be run on any environment. Supports macOS and Linux.
Decryption Key Provided	Requires the path to the data directory and the decrypted SQLCipher key. Can be run on any environment. Supports macOS and Linux.

7.1. Execution modes

`SignalForensics` offers three execution modes: Windows Automatic, Auxiliary Key Provided, and Decryption Key Provided. Table 14 presents each mode along with a brief description, indicating whether they are Windows-only and/or require execution in the same environment where Signal ran.

7.1.1. Windows automatic (`-m auto`)

This mode takes Signal's data directory as input and automatically decrypts the SQLCipher key. However, it requires execution within the same environment, including the same Windows account, where the Signal application was used. This mode decrypts the auxiliary key by calling the DPAPI directly through the `win32crypt` Python library, making it Windows only.

7.1.2. Auxiliary key provided (`-m aux`)

This mode takes Signal's data directory and the auxiliary key as inputs; it then uses the auxiliary key, as described in section 5.2.1, to obtain the SQLCipher key. Since these environment-dependent steps are skipped, this mode works for artifacts generated on all operating systems supported by Signal Desktop and can be executed in an external environment.

7.1.3. Decryption key provided (`-m key`)

This mode takes Signal's data directory and the SQLCipher key as inputs. Since all necessary inputs are provided by the user, this mode, like the Auxiliary Key Provided mode, works on external environments and for all Desktop platforms (Windows, macOS, and Linux).

8. Conclusion

This investigation had two primary objectives: decrypting Signal Desktop's locally stored data and analyzing its forensic artifacts in Windows environments. Our results confirmed that these artifacts contain valuable information, shedding light on the application's storage structure. To ensure reproducibility, we developed a detailed methodology and an automated script to streamline data extraction and forensic analysis operations on Signal Desktop. However, this process has limitations, particularly the need for access to DPAPI and the Windows user's credentials to complete the decryption of the SQLCipher key. Without this access, the auxiliary key must be manually extracted and provided as an input to the script. This requirement may restrict the applicability of the methodology in scenarios where access to the user's original environment — or to a virtual machine emulating it — is not possible. Despite these limitations, we consider this work a significant contribution to the field of digital forensic analysis. To date, no study has mapped the forensic artifacts present in Signal Desktop in such detail, nor has the Electron `safeStorage` mechanism been explored in the context of this application. Our findings provide a comprehensive guideline for forensic investigators, aiding in the interpretation of artifacts and facilitating the construction and validation of hypotheses during an investigation. Moreover, both the methodology and the developed script remain useful even in cases where encryption is not a barrier. The tool enables the examination of Signal's local data without opening the application, preventing potential alterations or contamination of files — an essential factor in preserving evidence integrity, particularly for expired messages that may still reside in the database until Signal is launched. Additionally, exporting results in CSV format ensures compatibility with various tools, allowing forensic investigators to manage and analyze the data flexibly.

For future work, we aim to develop an offline mode in `SignalForensics`, allowing it to bypass DPAPI using only the user's password, security identifier, and local DPAPI files, without requiring the original Windows system, enabling decryption in external environments. Furthermore, future research could explore the different security measures applied to the auxiliary key in other operating systems' environments, such as the Keychain in macOS, to assess the feasibility of this methodology across different environments.

CRediT authorship contribution statement

Gonçalo Paulino: Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Conceptualization. **Miguel Negrão:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Miguel Frade:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Patrício Domingues:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was partially by FCT/MCTES and EU funds under the project UIDB 04524/2020 and the project UIDB/EEA 50008/2020, Portugal.

Data availability

We have shared the link to our code in the Manuscript File's contents.

Message ID	Type	Conversation ID	Conversation Type	Conversation Name	Sent At	Received At	Author
01942477-cb71-7c66-8998-1ff2b705712e	outgoing	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:51:40 UTC	2025-01-06 17:51:42 UTC	Gonçalo
01942477-cb78-7abb-98f9-41d2292d3474	incoming	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:52:26 UTC	2025-01-06 17:52:28 UTC	Tiago Santos
01942477-cb7c-7dc1-bb8d-68d2b78857b3	incoming	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:52:32 UTC	2025-01-06 17:52:34 UTC	Tiago Santos
01942477-cb7e-7a3a-a184-02dd6a1af11a	outgoing	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:52:40 UTC	2025-01-06 17:52:40 UTC	Gonçalo
01942477-cb7f-74d3-9d8b-53485ee17e4d	incoming	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:52:37 UTC	2025-01-06 17:52:40 UTC	Tiago Santos
01942477-cb82-738c-897a-bfcdf18df719	outgoing	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:53:18 UTC	2025-01-06 17:53:20 UTC	Gonçalo
01942477-cb86-7bbc-8469-dec4c2cb07a7	outgoing	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:53:33 UTC	2025-01-06 17:53:35 UTC	Gonçalo
01942477-cb89-766f-947f-e42756ab385d	outgoing	49d68857-905f-43f4-bd37-10c3bf18c841	private	Tiago Santos	2025-01-06 17:53:52 UTC	2025-01-06 17:53:52 UTC	Gonçalo

Fig. 19. An example HTML report generated by SignalForensics (cropped to save space).

References

Abegaz, T., Phaxai, B., Payne, B., 2024. Investigating ephemeral messaging apps on iphone 14: insight into signal, whatsapp, snapchat, instagram, and confide. In: Arai, K. (Ed.), Proceedings of the Future Technologies Conference (FTC), vol. 4. Springer Nature Switzerland, Cham, pp. 312–331.

Abrams, L., 2024. Signal downplays encryption key flaw, fixes it after X drama. https://www.bleepingcomputer.com/news/security/signal-downplays-encryption-key-flaw-fixes-it-after-x-drama/.

Andriotis, P., Takasu, A., Tryfonas, T., 2014. Smartphone message sentiment analysis. https://doi.org/10.1007/978-3-662-44952-3_17.

Bilz, A., 2021. A forensic gold mine II: forensic analysis of signal messenger on windows 10. https://www.alexbilz.com/post/2021-06-07-forensic-artifacts-signal-desktop/.

Bowling, H., Seigfried-Spellar, K., Karabiyik, U., Rogers, M., 2023. We are meeting on Microsoft Teams: forensic analysis in Windows, Android, and iOS operating systems. J. Forensic Sci. 68, 434–460. https://doi.org/10.1111/1556-4029.15208.

Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D., 2020. A formal security analysis of the signal messaging protocol. J. Cryptol. 33, 1914–1983. https://doi.org/10.1007/s00145-020-09360-1.

Gupta, K., Lanka, P., Varol, C., 2024. A holistic digital forensic analysis of Discord – Storage, memory, and network perspectives. J. Forensic Sci. 69, 1320–1333. https://doi.org/10.1111/1556-4029.15548.

Heath, H., MacDermott, A., Akinbi, A., 2023. Forensic analysis of ephemeral messaging applications: disappearing messages or evidential data? Forensic Sci. Int. Digit. Investig. 46, 301585. https://doi.org/10.1016/j.fsidi.2023.301585. https://www.sciencedirect.com/science/article/pii/S266628172300094X.

Karo Karo, G.F.M., Hilal, M., Utomo, D.W., 2024. Analisis bukti digital aplikasi pesan terenkripsi signal desktop pada windows 11 menggunakan metode IDFIF v2. In: JRIIN: Jurnal Riset Informatika Dan Inovasi 1. http://jurnalmahasiswa.com/index.php/jriin/article/view/828 (Written in Indonesian, automated translation was used).

Kim, G., Hur, U., Kang, S., Kim, J., 2025. Analyzing the Web and UWP versions of WhatsApp for digital forensics. Forensic Sci. Int. Digit. Investig. 52, 301861. https://doi.org/10.1016/j.fsidi.2024.301861. https://www.sciencedirect.com/science/article/pii/S2666281724001884.

Kohn, M., Eloff, M., Eloff, J., 2013. Integrated digital forensic process model. In: Cybercrime in the Digital Economy. Comput. Secur. 38, 103–115. https://doi.org/10.1016/j.cose.2013.05.001. https://www.sciencedirect.com/science/article/pii/S0167404813000849.

Kret, E., Schmidt, R., 2024. Signal » specifications » the PQXDH key agreement protocol. https://signal.org/docs/specifications/pqxdh/.

Langley, A., Hamburg, M., Turner, S., 2016. Elliptic curves for security. RFC 7748, Internet Engineering Task Force. http://www.ietf.org/rfc/rfc7748.txt.

Madden, M., 2014. Public perceptions of privacy and security in the post-snowden era. https://www.pewresearch.org/internet/2014/11/12/public-privacy-perceptions/.

Marlinspike, M., 2016. WhatsApp's signal protocol integration is now complete. https://signal.org/blog/whatsapp-complete/.

NIST, 2001. FIPS 197. Advanced Encryption Standard. Technical Report. National Institute of Standards & Technology, Gaithersburg, MD, United States. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

OpenJS Foundation, Electron contributors, 2023. Build cross-platform desktop apps with JavaScript, HTML, and CSS | electron. https://www.electronjs.org/.

Perrin, T., Marlinspike, M., 2016. The double ratchet algorithm. Technical Report. Signal https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

Signal Foundation, 2025a. Group chats – signal support. https://support.signal.org/hc/en-us/articles/360007319331-Group-chats.

Signal Foundation, 2025b. Signal-Desktop/ts/model-types.d.ts at ca1d17354db10a14f9f5558dcb546af9f3bba578 · signalapp/Signal-Desktop. https://github.com/signalapp/Signal-Desktop/blob/ca1d17354db10a14f9f5558dcb546af9f3bba578/ts/model-types.d.ts#L148.

Signal Foundation, 2025c. Signal » documentation. https://signal.org/docs/.

Son, J., Kim, Y.W., Oh, D.B., Kim, K., 2022. Forensic analysis of instant messengers: decrypt signal, wickr, and threema. Forensic Sci. Int. Digit. Investig. 40, 301347. https://doi.org/10.1016/j.fsidi.2022.301347. https://www.sciencedirect.com/science/article/pii/S2666281722000166.

Steel, C., 2014. Idiographic digital profiling: behavioral analysis based on digital footprints. J. Digit. Forensics Secur. Law 9, 7–18. https://doi.org/10.15394/jdfsl.2014.1160.

Taneva, A.P., Smaragdakis, G., Picek, S., 2023. Reverse engineering of web cookies | TU delft repository. https://repository.tudelft.nl/record/uuid:8089d4d3-8e36-4b11-a9fc-17b43ebceb9a.

The Chromium Project, 2025. Chromium » home. https://www.chromium.org/chromium-projects/.

Turner, E., Rainie, L., Anderson, M., Perrin, A., Kumar, M., Auxier, B., 2019. Americans and privacy: concerned, confused and feeling lack of control over their personal information. https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/.

Zetetic, LLC, 2024. SQLCipher - full database encryption for SQLite. https://www.zetetic.net/sqlcipher/.