

## Bachelor-Thesis

# IT-forensische Analyse von SQLite Datenbanken mit kommerzieller und frei verfügbarer Software

eingereicht von:

Steffen Rapp

Studiengang IT-Forensik

Betreuer:

Prof. Dr.-Ing. Antje Raab-Düsterhöft

weitere Gutachter:

Prof. Dr. habil. Andreas Ahrens

Seeheim-Jugenheim, den 25. Juli 2024



## **Aufgabenstellung**

Die Bachelor-Thesis stellt eine immer wiederkehrende Frage der digitalen Forensik in den Mittelpunkt: Die Möglichkeiten zur forensischen Untersuchung und Analyse von SQLite-Datenbanken.

In der Bachelor-Thesis ist zu untersuchen, inwiefern Datensätze aus SQLite-Datenbanken wiederhergestellt und mit unter Umständen zur Verfügung stehenden temporären Dateien (Write-Ahead-Log / WAL) ergänzt werden können.

Dazu ist kommerzielle und frei verfügbare Software zu verwenden.

Es sind die Möglichkeiten zur Analyse und Wiederherstellung der verwendeten Software anhand von Szenarien zu untersuchen und zu bewerten.

Für die Evaluierung sind realitätsnahe SQLite-Datenbanken zu benutzen.

## Kurzreferat

SQLite-Datenbanken finden in einer Vielzahl von Anwendungen ihren Einsatz. Zu finden ist die „Database in a single file“ unter anderem in Browsern, Messengern, E-Mail-Programmen, Betriebssystemen und Frameworks. Somit dienen die Datenbanken als potenzielle Quelle von Beweisen für zum Beispiel Strafverfahren.

Daraus resultiert ein benötigtes Verständnis für die Beweisquelle und die Möglichkeit diese forensisch untersuchen zu können.

Diese Thesis verfolgt das Ziel zu untersuchen, wann gelöschte Datensätze aus SQLite-Datenbanken wiederherstellbar sind und welche Umstände dies begünstigen oder ausschließen.

Die praktische Umsetzung erfolgt anhand realitätsnaher SQLite-Datenbanken, die mit frei verfügbarer und kommerzieller Software untersucht werden.

Die gewonnene Erkenntnis, zeigt dass es unter bestimmten Umständen nachvollziehbar ist, wann Daten gelöscht, hinzugekommen oder geändert wurden und wann eine Wiederherstellung nicht mehr möglich ist.

**Abstract**

SQLite databases are used in a variety of applications. The "database in a single file" can be found in browsers, messengers, email programs, operating systems and frameworks, among others. The databases therefore serve as a potential source of evidence for criminal proceedings.

This results in a necessary understanding of the source of evidence and the possibility to examine it forensically.

This thesis aims to investigate when deleted data record from SQLite databases can be recovered and which circumstances favor or exclude this.

The practical implementation is based on realistic SQLite databases that are examined with opensource and commercial software.

The knowledge gained shows that under certain circumstances it is possible to trace when data has been deleted, added or changed and when recovery is no longer possible.

---

## Inhaltverzeichnis

1	Einleitung.....	9
1.1	Zielstellung.....	9
1.2	Abgrenzung.....	10
1.3	Aufbau der Arbeit.....	10
1.4	Problem.....	11
2	Forschungsstand.....	12
2.1	Stand der Wissenschaft.....	12
2.2	Stand der Technik.....	13
3	Grundlagen.....	15
3.1	Datenbanken.....	15
3.2	SQLite.....	16
3.2.1	Aufbau einer SQLite-Datei.....	17
3.2.2	B-tree-Seiten.....	19
3.2.3	Variable length integer (Varints).....	20
3.2.4	Struktur der Zelleninhalte.....	22
3.2.5	Freelists.....	24
3.3	Pragmas und deren forensische Relevanz.....	25
3.3.1	Journal_Mode.....	25
3.3.2	Rollback-Journal.....	26
3.3.3	Write-Ahead-Logs.....	28
3.3.4	Auto-Vacuum.....	30
3.3.5	Secure Delete.....	31
4	Material und Methoden.....	33
4.1	Forschungsumgebung.....	33
4.2	Software zur forensischen Analyse.....	34
4.2.1	Belkasoft Evidence Center X.....	34
4.2.2	Magnet Axiom.....	35
4.2.3	Autopsy / Sleuthkit.....	35
4.2.4	FQLite.....	36
4.3	Pythonskripts.....	37
4.3.1	Bring2lite.....	37
4.3.2	WAL Crawler.....	37
4.3.3	Wal2sqlite.....	37
4.3.4	Freeblock_seeking.....	38
4.4	Weitere Software.....	39
4.4.1	Visual Studio Code.....	39

---

4.4.2	DB Browser for SQLite .....	39
5	Ergebnisse Szenario Firefox .....	41
5.1	Belkasoft (Szenario Firefox).....	42
5.2	Axiom (Szenario Firefox) .....	47
5.3	Frei verfügbare Software (Szenario Firefox).....	50
5.3.1	Autopsy / Sleuthkit .....	50
5.3.2	FQLite .....	52
5.3.3	Pythonskripte .....	53
5.3.4	SQL-Abfragen mittels DB Browser for SQLite.....	54
6	Ergebnisse Szenario Memorix .....	57
6.1	Belkasoft (Szenario Memorix).....	57
6.2	Axiom (Szenario Memorix).....	61
6.3	Frei verfügbare Software (Szenario Memorix).....	64
6.3.1	Autopsy / Sleuthkit .....	65
6.3.2	FQLite .....	66
6.3.3	Pythonskripte .....	67
6.3.4	SQLite-Abfragen mittels DB Browser for SQLite .....	68
7	Vergleich der eingesetzten Software .....	69
7.1	Objektive Kriterien.....	69
7.2	Objektive Kriterien (Tabelle) .....	70
7.3	Bewertung der eingesetzten Software .....	71
7.3.1	Belkasoft .....	71
7.3.2	Axiom .....	71
7.3.3	Autopsy .....	71
7.3.4	FQLite .....	72
7.4	Zusammenfassung und Bewertung .....	73
8	Diskussion .....	74
8.1	Bewertung der Ergebnisse.....	74
8.1.1	Firefox .....	74
8.1.2	Memorix .....	77
8.2	Fazit .....	79
8.3	Ausblick.....	81
9	Anhang .....	83
9.1	Abfrage 1.....	83
9.2	Abfrage 2.....	83
9.3	Abfrage 3.....	83
9.4	Anhang A .....	83
9.5	Anhang Bewertungen .....	84

9.6	Freeblock_seeking.py .....	84
9.7	Wal2sqlite.py .....	86
	Literaturverzeichnis .....	91
	Abbildungsverzeichnis.....	96
	Tabellenverzeichnis.....	98
	Abkürzungsverzeichnis .....	99
	Selbstständigkeitserklärung .....	100



## 1 Einleitung

SQLite ist weltweit die am häufigsten genutzte relationale Datenbank [1]. Sie ist variabel einsetzbar und kommt unter anderem in Software zum Einsatz, die heutzutage die komplette Kommunikation und Informationsgewinnung eines Menschen abbildet.

So legen zum Beispiel die drei größten Instant-Messengerdienste Whatsapp, Facebook-Messenger [2] und WeChat [3] ihre Chatprotokolle in SQLite-Datenbanken ab. Allein Whatsapp hatte im Jahr 2022 2,4 Mrd. Nutzer weltweit [4].

Auch die drei großen Browserhersteller Chrome, Firefox und Safari [5](damit eingeschlossen deren Derivate oder Forks) speichern ihre Browserhistorie in SQLite-Datenbanken. Das gilt für die Desktopvarianten der Software sowie deren Mobilableger für Smartphones.

Laut Statista waren im April 2024 5,44 Mrd. Menschen mit dem Internet verbunden [6]. Google Chrome belegt mit einem Marktanteil von ca. 64% den ersten Platz [7].

Somit hat die forensische Auswertung von SQLite-Datenbanken eine bedeutende Relevanz als Beweismittel oder Erkenntnisbaustein, -gewinn. Es können Zeitabläufe und Aktivitäten nachvollzogen werden, Aussagen sind überprüfbar.

Daher ist die forensische Untersuchung von SQLite-Datenbanken ein Baustein für Strafverfolgungsbehörden, um Informationen zu gewinnen.

Diese Arbeit befasst sich mit den Möglichkeiten der Wiederherstellung von gelöschten Datensätzen, ob explizit oder durch die Programmroutine und wann eine Wiederherstellung nicht mehr möglich ist.

### 1.1 Zielstellung

Das Ziel der Arbeit besteht darin mit frei verfügbarer und kommerzieller Software zu untersuchen, unter welchen Umständen Datensätze wiederhergestellt werden

können und welche diese erschweren oder unmöglich machen.

Weiterhin versuchen wir über SQL-Abfragen weitere Informationen oder Fragmente zu extrahieren.

Abschließend werden wir einen Vergleich und Bewertung der eingesetzten Tools durchführen

## **1.2 Abgrenzung**

Zur Umsetzung dieser Thesis werden die folgenden Abgrenzungen formuliert:

- Es wird ausschließlich die Wiederherstellung und das Finden von Artefakten in einer SQLite-Datenbank und des dazugehörigen Write-Ahead-Logs betrachtet
- Nicht Gegenstand dieser Arbeit ist das Carven in einem Festplattenabbild von bereits gelöschten Inhalten eines WALs
- Ebenso ist es nicht Teil der Arbeit die programmier-technische Ablage von Datensätzen der Programme aufzuzeigen oder zu skizzieren
- Auch fließt die Verschlüsselung von Datenbanken durch SQLite nicht mit in diese Thesis ein
- Weiterhin bleibt die IT-forensische Arbeit mit Rollback-Journals außen vor, da wir keine aktuelle Software gefunden haben, die auf deren Verwendung setzt

## **1.3 Aufbau der Arbeit**

Wir ermitteln zuerst den Forschungsstand von Wissenschaft und Technik beziehungsweise der vorhandenen Software.

Danach gehen wir auf die Grundlagen von SQLite-Datenbanken ein und erklären die wichtigsten Einstellungen mit Bezug auf forensische Relevanz.

Anschließend stellen wir Material und Methoden vor, die in der Thesis zur Anwendung kommen und starten die forensische Untersuchung der SQLite-Datenbanken, welche in Festplattenabbildern hinterlegt sind.

Nach Abschluss der forensischen Untersuchung kommen wir zum Vergleich der

eingesetzten Software, bewerten die gefundenen Ergebnisse, ziehen ein Fazit und schließen mit einem Ausblick.

#### **1.4 Problem**

Im Vorfeld wurden die folgenden Probleme identifiziert:

Im Bereich der kommerziellen Software konnte in einer ersten Recherche festgestellt werden, dass SQLite-Datenbanken zur Auswertung unterstützt werden, allerdings ist der Umfang der Werkzeuge zur Analyse unklar.

In den Reihen von OpenSource-Software konnte die bekannte und frei verfügbare Software Autopsy mit der Unterstützung bei Analyse und Auswertung von SQLite-Datenbanken identifiziert werden. Erweitert durch zwei Plugins soll eine Wiederherstellung von gelöschten Einträgen möglich sein. Weiterhin wurden Pythonskripte für das Extrahieren von Daten aus SQLite-Datenbanken gefunden. Hier ist unklar, in welchem Umfang eine Analyse und Wiederherstellung durchführbar ist.

Nach Anfrage an einige Hersteller konnten wir nur für zwei kommerziell verfügbare Forensikprogramme eine Testlizenz erhalten.

Der Hersteller des vermeintlichen Goldstandards auf dem Gebiet von SQLite-Datenbanken, das Sanderson Forensic Toolkit, vergibt Lizenzen nur an Strafverfolger, Militär und Regierungen.

Ein weiterer Punkt ist die Erstellung beziehungsweise das Finden von SQLite-Datenbanken, welche sich in einem experimentellen Umfeld einsetzen und mit einem erträglichen Aufwand analysieren lassen.

## 2 Forschungsstand

### 2.1 Stand der Wissenschaft

Die forensische Analyse von SQLite-Datenbanken und Wiederherstellung von gelöschten Datensätzen in der Datenbank beziehungsweise in Verbindung mit Write-Ahead-Logs ist Thema in wenigen Veröffentlichungen.

Sangjeon [8] erläutert in seiner Arbeit den Aufbau von SQLite und analysiert Methoden zur Wiederherstellung von gelöschten Datensätzen in SQLite. Er wählt den Ansatz des Scannens einer Seite, um nach Abschluss die nicht zugeordneten Areale auf der Seite zu extrahieren.

2014 hat Shu [9] das Wiederherstellen von Datensätzen aus dem WAL untersucht. Er analysierte die Struktur einer WAL-Datei und entwickelte einen Algorithmus, um die von Firefox geschlossenen temporären Dateien aus einem Festplattenabbild mittels Frame-Splicing zu extrahieren, um dann Verlaufsdatensätze zu extrahieren.

Nachdem sich einige Tools auf dem forensischen Markt für die Analyse von SQLite-Datenbanken sammelten, entstand durch Nemetz [10] ein Korpus, um derartige Applikationen in einem Benchmark zur Vergleichbarkeit zu unterziehen. Das Benchmark sieht vor unterschiedliche Fehler (Encoding, Datenbankelemente, Baum-, Seitenstrukturen) in SQLite-Datenbanken einzubauen, um dann das Verhalten der Forensiksoftware zu prüfen.

2019 entwickelte Meng [11] das Pythonskript bring2lite. Das Skript soll die Extraktion von verwaisten Einträgen aus SQLite-Datenbank, Rollback-Journals und Write-Ahead-Logs beschleunigen und vereinfachen. Laut dem Artikel wurden von Forensikern häufig Kommandos wie „dd“, „file“ oder „strings“ zum Auffinden von Artefakten und deren Extraktion genutzt.

2023 wurde im Zuge der Arbeit von Daraghmi [12] ein Tool für das automatisierte Finden und Extrahieren von SQLite-Datenbanken auf Android-Geräten entwickelt. Die Arbeit vergleicht das Handling, Geschwindigkeit und die gefundenen Artefakte zwischen forensischen Tools (BelkaSoft, Magnet Axiom, Final Mobile und Forc).

## 2.2 Stand der Technik

Für die forensische Analyse von SQLite-Datenbanken haben laut Recherchen alle größeren kommerziellen Pakete eine Möglichkeit an Bord.

Allen voran geht das „Forensic Toolkit for SQLite“ von Paul Sanderson [13]. Es handelt sich hierbei um ein speziell auf SQLite-Datenbank-Forensik zugeschnittenes Werkzeug. Hiermit kann detailliert analysiert, wiederhergestellt und extrahiert werden. Es können Rollback-Journals sowie WALs in den Prozess mit einbezogen werden. Es gibt die Möglichkeit SQL-Abfrage mittels Query-Manager grafisch zu erstellen und Datenbankinhalte können zu bestimmten Checkpoint-Zuständen eines WALs angezeigt werden.

Belkasoft Evidence Center X [14] kann ebenso SQLite-Datenbanken und deren temporäre Journaldateien einlesen und innerhalb einer Tabelle anzeigen. Es werden Freelists und der nicht mehr zugeordnete Speicher nach Artefakten ausgelesen und diese aufbereitet.

Mit Version 3.1 hat Magnet Axiom [15] seinen SQLite Viewer erweitert. Es ist nun möglich in der Datenbank SQL-Abfragen abzusetzen. Auch sind diverse Manipulationen der Formatierungen für die einzelnen Spalten möglich (zum Beispiel anzeigen der Werte eine Spalte als Plist).

Laut eines Softwaretests des National Institute of Standards and Technology (NIST) [16] gibt es noch weitere forensische Softwarepakete die SQLite-Datenbanken interpretieren können. In diesen Tests wurde festgestellt, dass keine der getesteten Softwarepakete fehlerlos den Testparcour meistern konnte.

Bei FQLite handelt es sich um Opensource-Werkzeug [17] welches damit wirbt 100% im SQLite Forensic Corpus zu erreichen. Es handelt sich um ein grafisches Tool zur Analyse und Wiederherstellen gelöschter Einträge von SQLite-Datenbanken inklusive der temporären Journaldateien. Die neueste Version datiert von Mai 2024.

Des Weiteren gibt es noch frei verfügbare Pythonskripts, welche sich mit dem Finden, Wiederherstellen und Extrahieren von Datenbank-Einträgen beschäftigen:

Zum einen Undark welches im Benchmark Forensic Corpus getestet wurde. Dort

schnitt die Software nur durchschnittlich ab. Auch können nur Datenbanken, aber keine temporären Dateien analysiert werden. Allerdings ist es, laut Github-Seite [18] des Projekts, seit neun Jahren nicht mehr aktualisiert worden.

Zum anderen das bereits oben erwähnte bring2lite. Hier ist auch eine längere Abstinenz von Aktualisierungen festzustellen [19]. Seit circa fünf Jahren wird das Projekt nicht mehr weiterentwickelt. Der Vorteil besteht in der Unterstützung von Datenbanken und temporären Dateien (WAL sowie Rollback-Journals).

Zu guter Letzt ist noch ein kleines Skript aus dem Buch „Learning Python for Forensics Second Edition“ zu erwähnen. Der Sourcecode wird in Kapitel 12 [20] aufgeführt. Es kann nur Einträge aus einem Write-Ahead-Log in eine CSV-Datei extrahieren. Die Datenbank oder das Rollback-Journal werden nicht unterstützt.

### 3 Grundlagen

In diesem Kapitel werden technische Grundlagen erläutert, die für das Verständnis der Arbeit benötigt werden. Wir starten mit allgemeinen Grundlagen und gehen im weiteren Verlauf auf die technischen Details von SQLite ein.

#### 3.1 Datenbanken

Eine Datenbank vereinfacht den Zugriff auf einen Pool von darin abgelegten Daten, die untereinander eine logische Beziehung haben. Die Daten können über eine normierte Schnittstelle, dem DBMS, von einer Applikation gelesen, geschrieben oder für die Präsentation innerhalb der Applikation zusammengefasst werden. Dafür spricht die Applikation das DBMS an welche genaue Kenntnis über die Ablagestruktur der Daten hat. Die Schnittstelle wird immer gleich angesprochen, somit kann die Applikation ausgetauscht werden, ohne dass sich der Programmierer Gedanken über die Ablagestruktur der Daten machen muss. [21, pp. 1-15]

Bei SQLite handelt es sich um eine relationale Datenbank, die vom Transaktionsprinzip Gebrauch macht. Änderungen an den Datensätzen erfolgen zusammenhängend. Dabei können ein oder mehrere Datensätze beteiligt sein. Mit einem COMMIT schließt die Transaktion erfolgreich ab. Treten während der Transaktion unvorhergesehene Umstände auf, die es nicht ermöglichen die Transaktion vollständig durchzuführen, kann mittels Rollback-Befehl die Datenbank in ihren ursprünglichen konsistenten Zustand zurückgesetzt werden. Schicker [21, p. 18] beschreibt das von Härder und Reuter begründete ACID-Modell. Dieses wird auch im Zuge des SQLite-Journalbetriebs von Sanderson [22, p. 104] aufgegriffen. Transaktionen müssen:

- Atomar ausgeführt werden, es sollte nicht möglich sein nur einen Teil der Transaktion schreiben zu können
- Konsistent sein und nur Daten schreiben, die auch vorgesehen sind
- Isoliert sein, damit sich Änderungen nicht überschneiden
- Dauerhaft sein und nach einem Commit nicht verloren gehen

## 3.2 SQLite

Wie in der Einleitung beschrieben, handelt es sich bei SQLite um die weltweit am häufigsten genutzte relationale Datenbank. Die in C geschriebene Programmbibliothek, welche das eigentliche DBMS abbildet, ist gemeinfrei und der Sourcecode frei verfügbar. Seit dem Erscheinungsjahr 2000 wird die Programmbibliothek weiterentwickelt und ist aktuell in der Version 3.46.0 verfügbar [23].

Versionen sind für alle größeren Betriebssysteme (Windows, MacOS, Linux, Unix) verfügbar. Durch den freien Quellcode wäre auch eine Portierung auf andere Betriebssysteme möglich.

SQLite ist auch als „Database in a file“ bekannt und wurde ursprünglich für Embedded-Systeme entwickelt. Daher fehlt auch die Möglichkeit einer Rechteverwaltung innerhalb der Datenbank. Diese kann, sofern benötigt, nur über das Dateisystem abgebildet werden [5].

Zudem ist es möglich die Datenbank und deren Journaldateien „in memory“ abzulegen. Damit erstellen wir eine nicht-persistente Datenbank, die bei einem Neustart nicht mehr mit einfachen Mittel rekonstruiert werden kann.

Ansonsten bietet SQLite alles, was andere größere relationale Datenbanksysteme abbilden können. Von Abfragen via SQL über Transaktionen, Indizes, vorgefertigte Views bis zu Triggern können implementiert werden.

Das Betriebssystem Android bringt eine Implementierung der Programmbibliothek von SQLite mit. Werden spezielle Optionen innerhalb der Bibliothek benötigt, kann diese angepasst, kompiliert und in das Paket der Anwendung integriert werden. Solch eine Integration wird zum Beispiel im Browser Mozilla Firefox genutzt [24].

Die Größe der Bibliothek beträgt im Normalfall unter 1 MB (zwischen 590 KB und 750 KB) [25].

Die maximale Größe einer SQLite-Datenbank wird mit 281 Terabyte angegeben, einzelne Reihen haben ein Limit von einem Gigabyte [26].

SQLite-Datenbanken können entweder an ihrer typischen Endung (.sqlite3,



.sqlite, .db, .db3) erkannt werden oder per Magic Number (0x53514c69746520666f726d6174203300) die sich immer in den ersten 16 Byte der Datei befindet.

### 3.2.1 Aufbau einer SQLite-Datei

SQLite speichert Inhalte der Datenbank in Seiten ab. Alle Seiten innerhalb einer Datenbank besitzen immer die gleiche Größe. Diese ist im Header als Information abgelegt und beträgt zwischen 512 Byte und 65536 Byte. Jede Seite wird spezifisch dem Abbilden von bestimmten Informationen zugeordnet.

Seite Eins, welche auch als Root-Page bezeichnet wird, startet in den ersten 16 Bytes mit der Magic Number zur Identifizierung der Datei als SQLite-Datenbank (0x53514c69746520666f726d6174203300). Darauffolgend sind weitere 84 Byte, um Eigenschaften der Datenbank abzulegen. Von den insgesamt 23 Felder listen wir einige von Relevanz auf:

**Tabelle 3-1 Ausgewählte Header-Informationen einer SQLite-Datenbank-Datei [27]**

Offset	Größe (in Bytes)	Beschreibung
0	16	Header als Magic Number
16	2	Seitengröße der Datenbank (512 bis 65536 Byte)
18	1	Journal-Modus (Rollback-Journal oder Write-Ahead-Log)
24	4	Die Anzahl der Dateiänderungen
28	4	Die Größe der Datenbank in Seiten
36	4	Anzahl der Freelist-Seiten
56	4	Codierung innerhalb der Datenbank (UTF-8, UTF-16le, UTF-16be)

Am Ende der ersten Seite residiert die erste Tabelle der Datenbank, welche den Namen „sqlite\_master“ trägt. In dieser Tabelle sind alle CREATE-Statements zu

den in der Datenbank erstellten Tabellen, Indizes, Views und Triggers abgelegt. Als weitere Information sind die Root-Seiten der angelegten Tabellen aufgeführt.

Sind die Seiten für die abzubildenden Informationen zu klein, legt SQLite Overflow-Seiten an und speichert die restlichen Informationen dort ab.

Auf die Root-Seite folgen, vereinfacht dargestellt, vier Arten von Unterseiten:

- B-tree-Seiten (Tabellen oder Index)
- Overflow-Seiten
- Freelist-Seiten
- Pointer-Map-Seiten

„Der B-tree-Algorithmus ermöglicht die Speicherung von Schlüsseln und Daten mit eindeutigen und geordneten Schlüsseln auf seitenorientierten Speichergeräten.“ [27]

Die beiden B-tree-Arten teilen sich je nach Komplexität und Struktur der abzubildenden Datenbank in zwei Kategorien auf:

- interne Seiten
- Blattseiten

Interne Seiten sind vergleichbar mit einer Ansammlung von Zeigern, eine Art Index. Sie verweisen entweder auf eine oder mehrere weitere interne Seiten oder enden in Blattseiten.

Auf den einzelnen Blattseiten ist der eigentliche Datenbankinhalt in Spalten und Zeilen hinterlegt.

Overflow-Seiten können als Verlängerung einer B-tree-Seite gesehen werden. Ist eine Datenmenge so groß, dass diese nicht auf eine Seite passt, wird der Rest dorthin ausgelagert. Dieser Vorgang passiert z.B. bei abzuspeichernden BLOBs in Tabellen.

Freelist-Seiten sind eine Ansammlung von Seiten, die nicht mehr genutzt werden. Wir gehen in einem der folgenden Punkte näher darauf ein.

Pointer-Map-Seiten werden von SQLite zur Verwaltung genutzt. Dort sind die Verwendung und Zweck der einzelnen Seiten hinterlegt.

### 3.2.2 B-tree-Seiten

Auf B-tree-Seiten wird der eigentliche Inhalt der Datenbank, strukturiert in Tabellen als Zeilen, abgelegt. Die Seiten teilen sich in zwei mögliche Verwendungen auf. Zum einen für das Ablegen von Indizes und zum anderen für Daten. Da der Inhalt eines Index nur eine minimale forensische Relevanz besitzt, beschränken wir uns auf Verwendung für Tabellendaten [22, p. 43].

Jede B-tree-Seite besitzt den gleichen Aufbau. Sie startet mit einem Seiten-Header, zeigt über Pointer auf einzelne abgelegte Zellen und besitzt freie Blöcke innerhalb der Seite. Auch ist festzuhalten, dass jede Seite für genau eine spezifische Verwendung (Index oder Daten) und für genau eine spezifische Tabelle oder Index auftritt.

Zuvor ist zu erwähnen, dass jede Tabelle eine Rootseite besitzt. Diese verweist entweder auf interne oder Blattseiten. Ist es möglich den Inhalt der Tabelle auf eine Seite abzubilden, stellt die Rootseite gleichzeitig die erste und einzige Blattseite dar.

Der Header besteht aus acht oder 12 Bytes und enthält folgende Informationen:

**Tabelle 3-2 Aufbau eines B-tree Seitenheaders [27]**

Offset	Größe (in Bytes)	Beschreibung
0	1	Kennzeichnung des Seitentyps (interner Index / Tabelle, Blattseite für Index / Tabelle)
1	2	Adresse des ersten freien Blocks (> drei Bytes)
3	2	Anzahl der Zellen auf der Seite
5	2	Adresse des ersten gültigen Eintrags
7	1	Gesamtzahl in Bytes des freien Speichers (< vier Bytes)
8	4	Den Right-most pointer (nur relevant für interne B-tree-Seiten)

Um nun die einzelnen Zelleninhalte auf einer Seite zu finden kann wie folgt vorgegangen werden. Man identifiziert den Seitentyp (in unserem Fall eine Blattseite 0x0D) und prüft Offset drei und vier für die Anzahl der Zellen. Auf den Header folgt in zwei Byte-Schrittfolgen der Offset für einzelne Zelleninhalte.

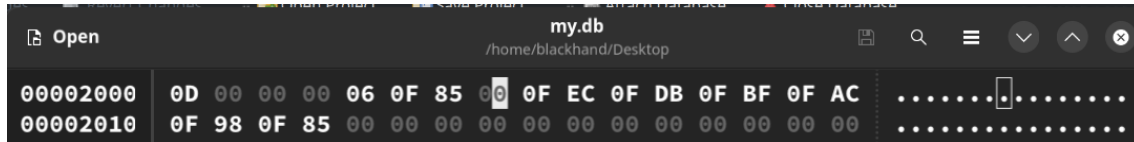


Abbildung 3-1 Abbildung eines B-tree-Headers

Als Beispiel sehen wir in dieser Abbildung in Offset Null die Identifizierung der Seite als B-tree Blattseite (0x0D). Die nächsten zwei Bytes geben an, dass kein freier Block zu finden ist. Danach folgt die Anzahl der Zeilen (0x0006), gefolgt vom Offset-Wert die uns innerhalb der Seite zur ersten Zeile führt (0x0F85). Abschließend (0x00) existiert kein fragmentierter Speicherplatz. Die folgenden sechs mal zwei Bytewerte stellen die Offsets für die einzelnen Zeilen auf dieser Seite dar (0x0FEC, 0x0FDB, 0x0FBF, etc.).

Sofern der Wert für die Adresse des ersten freien Blocks nicht Null ist, führt diese Adresse zum Offset des ersten freien Blocks. Dieser Block startet mit einer zwei mal zwei Bytes großen Adresse. Die ersten zwei Bytes geben den darauffolgenden freien Block an, gefolgt von der Länge in Bytes des aktuellen Blocks. Bei freien Blöcken handelt es sich um gelöschte Zeilen. Hier können Informationen wiederhergestellt werden, allerdings ist darauf zu achten, dass die Payloadlänge und oder die ROWID nicht mehr eindeutig zu rekonstruieren sind, da sie vom Header des freien Blocks teilweise überschrieben wurden.

### 3.2.3 Variable length integer (Varints)

Bevor die Struktur der Ablage von Daten besprochen wird, zeigen wir auf wie Werte für die Payloadlänge und ROWID von SQLite sparsam mittels Varints abgelegt werden. Dabei handelt es sich um „eine statische Huffman-Kodierung von 64-Bit-Ganzzahlen mit Zweierkomplement“ [27]. Dies gilt für kleinere positive Zahlen, sobald es sich um negative Zahlen handelt, steigt der Speicherverbrauch. Allerdings sind Payloadlänge und ROWID überwiegend als positive Zahl gespeichert.

Über Varints können 64 Bit-Zahlen abgebildet werden. Die Länge variiert zwischen einem und maximal neun Byte.

Ob das nächste Byte in der Reihenfolge zum Varint zählt, entscheidet das gesetzte oder nicht gesetzte MSB. Somit kann für die ersten acht Bytes eines Varints festgehalten werden, ist der Wert über 0x7F wird das folgende Byte benötigt. Handelt es sich um das letzte und somit neunte Byte fließen alle 8 Bit in die Rekonstruktion mit ein. Das aktuelle Byte wird in die binäre Form umgerechnet und das MSB abgeschnitten. Die Bitfolgen werden verkettet und ergeben so den ursprünglichen Wert.

Als Beispiel dient der folgende Ausschnitt:

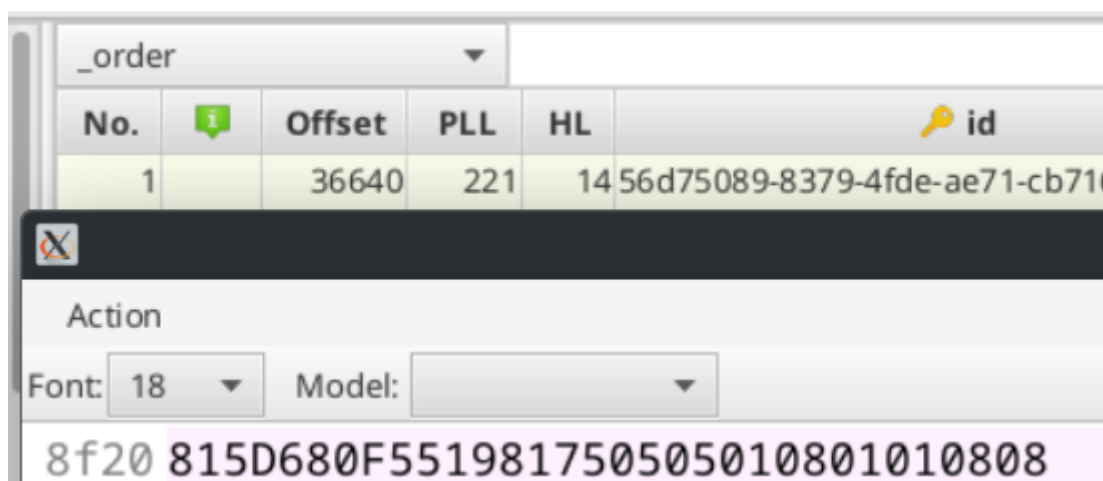


Abbildung 3-2 Ausschnitt eines Datensatzes mit hexadezimaler Darstellung der ROWID

Das obere Fenster bildet einen Datensatz ab, welcher in der später besprochenen „notes\_db“-Datenbank enthalten ist. Das untere Fenster zeigt diesen Eintrag im Hexformat. Die Payloadlänge (PLL = 211) wird als Varint abgebildet und steht am Anfang eines Zelleneintrags.

0x81 in binär 10000001, MSB ist gesetzt. Wir übernehmen die letzten sieben Bit und beziehen das nächste Byte ein.

0x5D in binär 01011101, MSB ist nicht gesetzt. Wir übernehmen die letzten sieben Bit und beziehen kein weiteres Byte mit ein.

Wir verketteten nun die beiden Werte 00000001 und 1011101 und erhalten in Dezimalschreibweise 211, Hexadezimal 0xDD.

Die Speicherersparnis ist dann sichtbar, sofern man den Wert als 64-Bit-Wert

abspeichern möchte: 0x0000000000000000DD. [28] [22, p. 26]

### 3.2.4 Struktur der Zelleninhalte

Die Struktur der abgelegten Daten ist von Seitenformat zu Seitenformat leicht unterschiedlich. Meist sind drei zentrale Werte vorhanden: Payloadlänge, Payload und ein Zeiger auf eine Overflow-Seite. Wir schauen uns das Datensatzformat für eine B-tree-Blattseite an. Auf ihr sind die meisten Informationen zu finden.

Jeder Datensatz startet mit der Gesamtlänge des Payloads, welcher als Varint formatiert ist. Danach wird die ROWID oder ein Integer Key, ebenso im Format eines Varints, abgebildet, um Zeilen innerhalb einer Tabelle eindeutig identifizierbar zu machen. Der Integer Key kommt bei Tabellen ohne ROWID zum Einsatz. Handelt es sich um eine übliche ROWID-Tabelle steht der Wert gleichzeitig für den Integer Key. Im SQLite-Datenbankkontext steht der Integer Key als Synonym für einen automatisch inkrementierenden Primärschlüssel.

Die beiden genannten Werten können auch als Zellen-Header aufgefasst werden. Darauffolgend wird der eigentliche Payload abgebildet und abschließend ein vier Bytes großer Zeiger auf eine Overflow-Seite, allerdings nur wenn diese auch belegt wird.

Der Abschnitt Payload ist in zwei Abschnitte zu teilen. Einmal „Record-Header“ gefolgt von den eigentlichen Daten. Der Kopf des Eintrags besteht aus den Informationen der Länge in Bytes als Varint gefolgt von den Datentypen der einzelnen Spalten. Im Anschluss folgen die einzelnen Werte für die Spalten der Zeile.

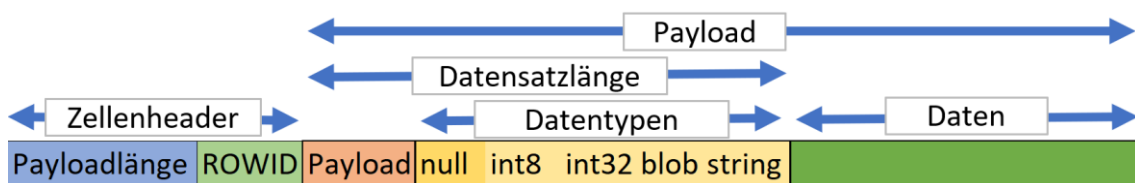


Abbildung 3-3 Schematischer Aufbau eines Datensatzes in einer B-tree-Blattseite [22, p. 30]

Bei der Angabe der Payloadgröße ist darauf zu achten, dass der Wert für die Größe mit einfließt. Im Abschnitt des Payload sind für die Spalten vorgesehene Datentypen hinterlegt.

Tabelle 3-3 Auflistung der Codierung für Datentypen [27]

Datentyp	Größe	Beschreibung
0	0	Wert ist Null
1	1	8 Bit Integer
2	2	16 Bit Integer
3	3	24 Bit Integer
4	4	32 Bit Integer
5	6	48 Bit Integer
6	8	64 Bit Integer
7	8	64 Bit Gleitkommazahl nach IEEE754-2008
8	0	Wert des Integer ist 0
9	0	Wert des Integer ist 1
10,11	variabel	Reserviert für internen Gebrauch
N>12	$(N-12)/2$	Wert ist ein Blob von $(N-12)/2$ Byte wobei N gerade ist
N>13	$(N-13)/2$	Wert ist ein String von $(N-13)/2$ Byte wobei N ungerade ist

Um dies zu visualisieren, folgt eine beispielhafte Beschreibung:

Bei dem folgenden Ausschnitt handelt es sich um die Tabelle Mitarbeiter einer Beispieldatenbank. Das „CREATE TABLE“-Statement haben wir aus der „sqlite\_master“-Tabelle entnommen:

Tabelle 3-4 gekürzter Ausschnitt des Create Table-Statements der Tabelle Mitarbeiter

```
mitarbeiter_id INTEGER PRIMARY KEY,
abteilungid INT NOT NULL,
strasse TEXT NOT NULL,
ort_id INT NOT NULL,
```

```
raumid INT NOT NULL,
nachname TEXT NOT NULL,
vorname TEXT NOT NULL,
```

Nun folgt der zu interpretierende Datensatz:

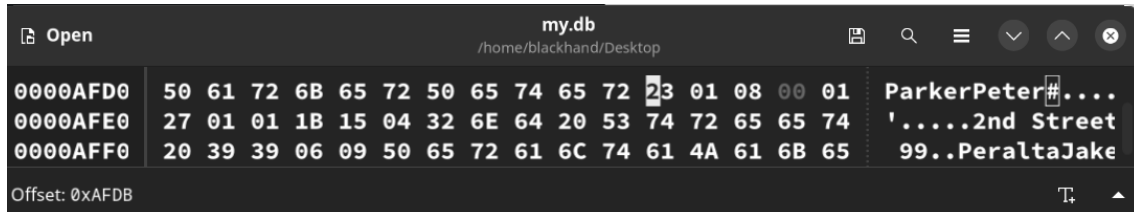


Abbildung 3-4 Beispieldatensatz einer Zeile in einer SQLite-Datenbank

Der Datensatz startet mit dem markierten Offset 0xAFDB. Die Payloadlänge beträgt 0x23 (dezimal 35 Bytes), 0x01 ist die ROWID des Eintrags. Darauf folgt mit 0x08 die Länge des Payloads der Zeile. Die Datentypen der Zeile lauten wie folgt:

0x00 stellt die erste Spalte und damit den Primärschlüssel dar. Der Wert ist Null, da in diesem Fall die ROWID stellvertretend für den Primärschlüssel steht.

0x01 steht für einen 8 Bit Integer-Wert und stellt die „abteilungsid“ dar (0x04).

0x27 steht für einen ungeraden Dezimalwert von 39. Damit handelt es sich um einen 13 Bytes großen String und repräsentiert den Wert für die „strasse“ (0x32 6E 64 20 53 74 72 65 65 74 20 39 39).

0x01 steht für einen 8 Bit Integer-Wert und bildet die „ort\_id“ ab (0x06).

0x01 steht für einen 8 Bit Integer-Wert und bildet die „raumid“ ab (0x09).

0x1B steht für einen ungeraden Dezimalwert von 27. Damit handelt es sich um einen 7 Bytes großen String welcher den Wert für die Spalte „nachname“ (0x50 65 72 61 6C 74 61) repräsentiert.

0x15 steht für einen ungeraden Dezimalwert von 21. Damit handelt es sich um einen 4 Bytes großen String welcher den Wert für die Spalte „vorname“ (0x 4A 61 6B 65) repräsentiert.

### 3.2.5 Freelists

Freelists verhalten sich zu Freeblocks ähnlich. Hier spricht man aber nicht von



einer kleineren Bytefolge auf einer Seite, sondern von der ganzen Seite an sich, die aktuell nicht mehr genutzt wird.

Im Header-Feld der Root-Seite ist der Wert für die erste Seite der Freelists hinterlegt. Springt man zu dieser Offset-Adresse, erreicht man eine Trunkpage, in welcher alle aktuell zur Verfügung stehenden freien Seiten verzeichnet sind. Sofern nicht alle Seiten auf einer Trunk-Page verzeichnet werden können, bilden die ersten vier Bytes der Seite die Verlinkung zur nächsten Trunkpage. Die nächsten vier Bytes zeigen die Anzahl der auf dieser Seite verlinkten freien Seiten. Die folgenden Bytewerte (immer vier) bilden die Seitenzahlen ab.

### 3.3 Pragmas und deren forensische Relevanz

SQLite unterstützt in der aktuellen Version (3.46) 66 PRAGMA-Optionen, die das Verhalten der Datenbank beeinflussen können. Diese können teilweise bei der Kompilierung der Programmbibliothek oder zur Laufzeit der Applikation für die Datenbank gesetzt werden. Allerdings sind nur wenige forensisch relevant.

In diesem Abschnitt erläutern wir die forensische Relevanz des „journal\_mode“, „secure\_delete“ und „auto-vacuum“.

#### 3.3.1 Journal\_Mode

Da es sich, wie bereits weiter oben erwähnt wurde, bei SQLite um eine transaktionale Datenbank handelt, müssen alle Änderungen dem beschriebenen ACID-Prinzip folgen.

Das Pragma Journal\_Mode sieht dazu drei Optionen vor:

- Rollback-Journals arbeiten mit den Modi „DELETE“, „TRUNCATE“ und „PERSIST“
- Für das Write-Ahead-Log ist die Option WAL zu wählen
- „MEMORY“ schreibt Änderungen in den flüchtigen Arbeitsspeicher
- Die Option „OFF“ schaltet den journal\_mode ab

Bei den Modi „MEMORY“ und „OFF“ besteht eine hohe Wahrscheinlichkeit, dass Transaktionen bei unvorhergesehenen Umständen (Programmabsturz, Betriebssystemcrash, Stromverlust) nicht rückgängig gemacht werden können,

die Datenbank korrumpiert und danach nicht mehr zu verwenden ist. Das ACID-Prinzip wird mit dieser Einstellung nicht verfolgt beziehungsweise ausgesetzt.

Die Modi für das Rollback-Journal sind seit Start der Entwicklung von SQLite verfügbar und geben an wie mit dem Inhalt des Journals bei neuen Transaktionen verfahren wird.

Die Write-Ahead-Logs wurden mit Version 3.7.0 (2010) eingeführt und hat im Gegensatz zu den Rollback-Journals einige Vorteile. WALs arbeiten immer im Modus „persistent“.

Der Journal-Modus kann in einer bestehenden Datenbank geändert werden allerdings nicht, sofern eine Transaktion aktiv ist.

### 3.3.2 Rollback-Journal

Unter Rollback-Journal sind die Optionen „DELETE“, „TRUNCATE“ und „PERSIST“ zu verstehen. Mit ihnen wird festgelegt, was mit dem Inhalt eines bestehenden Journals geschieht.

Zuvor beleuchten wir die Arbeitsweise des Rollback-Journals und zeigen die forensische Relevanz.

Wird eine Transaktion gestartet, legt SQLite eine temporäre Datei an. Diese wird im gleichen Verzeichnis der Datenbank mit dem Suffix „-journal“ erzeugt. In dieser Datei werden mit einer bestimmten Struktur die Seiten der Datenbank abgelegt auf welchen eine Änderung erfolgen soll.

Der Aufbau der Journal-Datei ist unkompliziert. Sie startet mit einem Header gefolgt von der Seitennummer, Seiteninhalt und einer Checksumme für diese Seite.

Der Header enthält die folgenden Informationen:

- Signatur der Datei beziehungsweise die Magic Number in acht Bytes (0xd9d505f920a163d7)
- Die Seitenanzahl in diesem Journal (vier Bytes)
- Eine zufällige Nonce zur Berechnung der Checksumme einzelner Seiten (vier Bytes)
- Seitenanzahl in der Datenbank bei Aktualisierung des Journals (vier

Bytes)

- Größe eines Sektors auf der Festplatte (vier Bytes)
- Anzahl der Seiten im Journal (vier Bytes)
- Gefolgt von einem Auffüllen durch Nullen auf die Größe eines Sektors der Festplatte

Auf den Header folgt dann die Seitennummer (vier Bytes), der Seiteninhalt (die Größe der Seite ist abhängig von der in der Datenbank gewählten Seitengröße) und abschließend der Checksumme.

Die Checksumme wird durch das Addieren von Werten auf den Wert der Nonce generiert. Zuerst wird ein Index aus der Seitengröße (zum Beispiel 1024 Bytes) subtrahiert mit 200 generiert. Dieser Indexwert wird als Offset interpretiert, mittels welchem das Byte an diesem Offset der Seite extrahiert wird. Diese Byte wird als acht Bit Integer ohne Vorzeichen auf die Nonce addiert.

Der Index wird so lange mit 200 subtrahiert bis der Wert kleiner oder Null entspricht. Ist dies der Fall, ist die zuletzt berechnete Nonce die Checksumme der Seite im Rollback-Journal.

Nachdem wir den generellen Aufbau eines Rollback-Journals besprochen haben, kommen wir nun zur Arbeitsweise:

Sollen Änderungen aufgrund einer Transaktion in der Datenbank durchgeführt werden, schreibt SQLite die betroffenen Seiten in das Journal. Zuvor wird ein neuer Header inklusive Nonce erzeugt. Somit können pro Seite die Checksummen erstellt werden.

Die neuen Werte werden geschrieben und im Idealfall die Transaktion abgeschlossen, der Cache geleert und das Journal als ungültig markiert. Tritt ein Fehler bei der Transaktion auf, schreibt SQLite die gültigen Seiten aus dem Journal zurück in die Datenbank.

Wie zu Anfang dieses Punktes beschrieben gibt es drei Modi für Rollback-Journals. Anhand dieser Modi wird festgelegt, wie SQLite das Journal ungültig markiert.

- „DELETE“ löscht die Journal-Datei, nachdem eine Transaktion erfolgreich abgeschlossen wurde.

- „TRUNCATE“ schneidet den Inhalt unterhalb des Headers „ab“ beziehungsweise füllt mit Null auf. Somit muss keine neue Datei bei jeder Transaktion erstellt werden.
- „PERSIST“ ist forensisch gesehen der beste Modus des Rollback-Journals. Ist eine Transaktion erfolgreich, wird nur der Header des Journals gelöscht. Die vorher ausgelagerten Seiten bleiben im Journal bestehen.

Zuletzt ist festzuhalten, dass laut Veröffentlichungshistorie die grundlegende Struktur und das Verhalten des Rollback-Journals seit Implementierung nicht angepasst worden sind.

### 3.3.3 Write-Ahead-Logs

Für Write-Ahead-Logs bleibt nur eine Option im Pragma „journal\_mode“ die mit „WAL“ angegeben ist. WAL ist eine Alternative zu Rollback-Journals die den Vorgang der Aktualisierung bei einer Transaktion umdreht.

Auch hier schauen wir uns das Vorgehen und die forensische Relevanz des Modus an.

Sofern eine Transaktion gestartet wird, wird eine temporäre Datei mit dem Namen der Datenbank und dem Suffix „-wal“ angelegt. In diese Datei werden alle vorzunehmenden Änderungen eingetragen. Das heißt der Wert, welcher geändert werden soll, wird in der Datenbank gesucht, die Seite mit diesem Wert in den WAL übertragen und die Änderung vorgenommen. Die geänderten Seiten bleiben so lange im WAL, bis eine Bedingung für einen Checkpoint eintritt:

- Der WAL erreicht die Größe einer definierbaren Seitenanzahl (Standardwert ist 1000 Seiten)
- Der Nutzer initiiert einen Checkpoint
- Das Programm, welches SQLite verwendet, wird beendet / geschlossen

Bis zum Eintritt des Checkpoints werden keine Änderungen an den Daten der Datenbank vorgenommen.

Zudem existieren in diesem Fall zwei Headerformate. Das erste Format beschreibt den allgemeinen Dateiheader des WALs. Der zweite Header

umschließt die einzelnen Frames.

Der Datei-Header eines WALs besteht immer aus 32 Bytes und wird im Gegensatz zum Rollback-Journal nicht mit Null auf eine Sektorengröße aufgefüllt. Er besteht aus den folgenden Informationen:

- Der Dateisignatur, die entweder den Wert 0x377f0682 oder 0x377f0683 annimmt (der gewählte Wert hat Einfluss auf die Art der Berechnung der Checksumme)
- Der Dateiformat-Version (aktuell 3007000)
- Der Seitengröße der Datenbank (zum Beispiel 1024 Bytes)
- Checkpoint-Sequenznummer, welche bei jedem Checkpoint um 1 inkrementiert wird (startet bei 0)
- Salt 1 ist ein zufällig gewählte Integer der bei jedem Checkpoint um 1 erhöht wird
- Salt 2 ist ein weiterer zufällig gewählter Integer der sich bei jedem Checkpoint ändert
- Checksumme 1
- Checksumme 2

Auf den Datei-Header folgen Null oder mehrere WAL Frame Header. Ein Frame innerhalb eines WALs umfasst eine zu ändernde Seite der anliegenden Transaktion. Er ist wie folgt aufgebaut:

- Die Seitenzahl der Seite aus der Datenbank
- Der Indikator für einen Commit (Wert 0) oder die Anzahl der Seiten in der Datenbank nach der Transaktion
- Salt 1 und 2 aus dem Datei-Header (jeweils 4 Byte)
- Checksumme 1 kumulativ auf Basis der vorhergehenden und einschließlich der aktuellen Seite
- Checksumme 2 als zweiter Teil der kumulativen Checksumme 1

Zu erwähnen ist noch der Shared Memory File (zu erkennen am Namen der Datenbank mit dem Suffix -shm). Er hat allerdings keinerlei forensische Relevanz und wird daher nicht besprochen.

Der Verlauf bei der Nutzung eines WALs von SQLite gestaltet sich wie folgt. Die

Transaktion startet und die zu ändernden Seiten werden aus der Datenbank in den WAL kopiert, einzelne Werte hinzugefügt, geändert, gelöscht. Ist die Transaktion beendet wird im letzten WAL Header Frame der Wert für Commit auf 0 gesetzt. Die nächste Transaktion startet und weitere Seiten werden im WAL an die schon bestehenden Seiten angehängt. Ein Checkpoint wird ausgelöst und somit werden die neuesten Kopien einzelner Seiten in die Datenbank übertragen.

Startet nach diesem Checkpoint eine neue Transaktion werden im Datei-Header die Sequenznummer und Salt 1 um 1 erhöht. Salt 2 bekommt einen neuen zufälligen Wert zugeordnet, die Checksummen werden neu berechnet. Die zu ändernden Seiten werden erneut in den WAL ausgelagert und überschreiben die vorhandenen Inhalte. Die alten Inhalte werden nicht gelöscht.

Erst wenn die Applikation ordentlich beendet wird, erzeugt das einen letzten Checkpoint und der WAL wird gelöscht.

Über das Vorgehen von SQLite im „journal\_mode“ WAL lässt sich die forensische Relevanz feststellen. Durch das Überschreiben vorhandener Einträge ist es möglich aus vorherigen Checkpoints, Seiten im Write-Ahead-Log in einer anderen Version als sie in der Datenbank vorhanden sind zu extrahieren. Es ist also unter idealen Umständen möglich eine Änderungshistorie einzelner Tabelleninhalte nachzuvollziehen oder eben gelöschte / geänderte Werte in ihren Ursprung zurückzusetzen.

Auch ist festzuhalten, dass die grundlegende Struktur eines Write-Ahead-Logs seit Einführen nicht geändert worden ist. Trotz der im Header befindlichen Versionsnummer konnten wir in der Recherche keinen Anhaltspunkt auf eine Änderung des Verhaltens oder der Struktur finden.

### **3.3.4 Auto-Vacuum**

Diese Option kann in drei möglichen Modi arbeiten:

- 0 = NONE
- 1 = FULL
- 2 = INCREMENTAL

Im Modus „NONE“ ist die Funktion ausgeschaltet. Die Datenbank wird nach

Löschen von Einträgen nicht kleiner. Seiten, die nicht mehr genutzt werden, sind als „Freelist“ markiert und werden für zukünftige einzufügende Daten genutzt.

„FULL“ veranlasst SQLite dazu „Freelist“-Seiten an das Ende der Datenbank zu verschieben, wo sie dann nach einem erfolgreichen „Commit“ einer Transaktion abgeschnitten werden.

Für den letzten Modus muss ein zusätzliches Pragma („incremental\_vacuum(N)“) konfiguriert werden. Die zu setzende Variable N gibt an, wie viele Seiten in „Freelists“ auflaufen müssen bevor dort abgeschnitten wird.

Sofern bei einer forensischen Untersuchung die Journaldateien mitgesichert werden konnten kann dieses Pragma im Idealfall umgangen werden und hat keine Auswirkung auf die Wiederherstellung von Daten aus der Datenbank.

### **3.3.5 Secure Delete**

Das Pragma wurde im April 2006 (Version 3.3.5) eingeführt und kann, sofern vorhanden, über gesetzte Journal-Modes übergangen werden. Die Option ist beim Kompilieren der Programmbibliothek standardmäßig „OFF“.

Mit der Version 3.6.12 aus März 2010 ging die Option als fester Bestandteil von SQLite in die Programmbibliothek über, sie war somit ein fester Bestandteil der Kompilierung und musste explizit ausgeschaltet werden.

Wird ein Eintrag in der SQLite-Datenbank gelöscht wird dieser als „Freeblock“ markiert. Sofern die genannte Option ausgeschaltet ist, werden die ersten vier Bytes des Eintrags auf Null gesetzt. Somit kann der Eintrag mit einem kleinen Informationsverlust wiederhergestellt werden.

Ist „secure\_delete“ eingeschaltet, überschreibt SQLite den kompletten Eintrag mit Null. Er ist danach nicht mehr wiederherzustellen.

Im August 2018 wurde eine dritte Option, „FAST“, zu „secure\_delete“ hinzugefügt. Dieser Modus überschreibt den Eintrag nur dann mit Null sofern es keinen Anstieg im I/O-Bereich der Datenbank gibt. SQLite [29] postuliert ein Verschwinden von forensischen Spuren in der B-Tree-Seite, aber nicht auf den Freelist-Seiten.

Sofern die betroffenen Seiten noch ungeändert in der Datenbank stehen

(Nutzung von WALs) oder die Seiten im Rollback-Journal vorhanden sind, kann der ursprüngliche Wert mittels forensischer Untersuchung wiederhergestellt werden.



## 4 Material und Methoden

In diesem Punkt gehen wir auf unsere Forschungsumgebung und die Generierung der Festplattenabbilder ein. Auch erläutern wir die von uns genutzte Software zur forensischen Analyse der Datenbanken.

### 4.1 Forschungsumgebung

Die Umsetzung der Analyse erfolgte in einer hybriden Umgebung. Die Daten für Szenario Eins von Firefox (Version 127.0) wurden in einer virtualisierten Windows 10-Maschine (Version 22H2 Build 19045.3803) mittels Virtualbox in der Version 7.0.10 r158379 erzeugt. Welche Webseiten aufgerufen wurden, ist Anhang A zu entnehmen.

Um eine bessere Nachvollziehbarkeit zu gewährleisten, wann Firefox welche Daten in seine Datenbank schreibt, arbeiten wir mit zwei Festplattenabbildern der virtuellen Maschine.

Das erste Abbild wurde vor dem Löschen des Browserverlaufs gesichert, das zweite, nachdem der Browserverlauf gelöscht wurde. Beide Abbilder liegen im VDI-Format vor.

Für Szenario Zwei wurde die Datenbank der Android-App Memorix Notizen + Checklisten“ (ID im Appstore: „panama.android.notes“) genutzt. Die Datenbank entstammt aus einem nicht mehr genutzten Smartphone des Verfassers.

Da das Smartphone mit einem Custom-Rom ausgestattet, aber nicht gerootet ist, konnte kein Android-Backup oder unverschlüsseltes Abbild des internen Speichers mittels forensischer Software erstellt werden.

Allerdings konnten die Daten mittels „adb root“-Befehl über die Android Debug Bridge extrahiert werden. Da es in dieser Arbeit primär um die Forensik und Wiederherstellung von gelöschten Datensätzen geht, wurde die Datenbank inklusive der noch vorhandenen WAL auf den Desktop der virtualisierten Windows 10-Maschine abgelegt.

Die Software Belkasoft Evidence Center X konnte nicht in einer virtuellen

Maschine installiert werden, daher läuft die Software auf dem Host des Verfassers.

Magnet Axiom konnte in einer virtuellen Windows 10-Maschine (Version 22H2 Build 19045.4529) installiert werden.

Autopsy konnte stabil unter einer virtuellen Linux-Maschine mittels EndeavourOS-Distribution (Build Gemini Gemini-2024.04.20, Kernel 6.9.5) installiert werden.

Um mittels Autopsy die Festplattenabbilder forensisch zu untersuchen, mussten diese mittels FTK Imager (Version 4.7.1.2) der Firma AccessData vom vorliegenden VDI-Format in RAW umkonvertiert werden.

## **4.2 Software zur forensischen Analyse**

### **4.2.1 Belkasoft Evidence Center X**

Das Evidence Center X von Belkasoft ist eine forensische Software zum Untersuchen und Erstellen von Datenträgerabbildern von physischen Laufwerken. Laut Webseite operiert das Unternehmen in 130 Ländern und unterstützt mit seiner Software Strafverfolger, Unternehmen, Regierungen und Hochschulen / Universitäten.

Der Einsatzbereich erstreckt sich von digitalen forensischen Untersuchungen über Cyber Response bis hin zu eDiscovery.

Es können alle gängigen Betriebssysteme analysiert und eingelesen werden.

Unterstützt werden bereits erstellte Datenträgerabbilder sowie die Erstellung dieser von Dronen, Mobilgeräten (Android, iOS) oder physischen Festplatten.

Nach Auswahl des gewünschten Abbilds kann angegeben werden welche Artefakte im Abbild gesucht werden sollen. Hier steht eine reichhaltige Auswahl aller möglichen Browser-, Social Media- und Dateityp-Artefakte zur Verfügung.

Ist das Image vollständig durchsucht können einzelne Dateien mit integrierten Viewern manuell untersucht werden.

Laut Webseite ist auch eine detaillierte Analyse von SQLite-Datenbanken

inklusive des WALs möglich.

Die Software war für uns per Testlizenz unter Windows zugänglich. Wir nutzen zu unserer Analyse die Version 2.5.

#### **4.2.2 Magnet Axiom**

Magnet Axiom wird, wie die Software von Belkasoft, zur Erstellung und forensischen Analyse von Datenträgerabbildern eingesetzt.

Magnet Forensic existiert seit 2011 und hat kontinuierlich seine Reichweite und Kunden ausgebaut. Laut Webseite zählt man 4000 Kunden in 100 Ländern. Es werden Strafverfolger, Unternehmen sowie Universitäten bedient.

Auch in Magnet Axiom ist es möglich bereits existierende Festplattenabbildern einzulesen oder diese zu erstellen. Es unterstützt bei der Erstellung sowie Untersuchung aller gängigen Betriebssysteme.

Weiterhin gibt es eine Vielzahl an Artefakt-Optionen und die Möglichkeit der Einbindung von künstlicher Intelligenz zur Analyse von Artefakten (z.B. Klassifikation von Bildern).

Nachdem ein Abbild durchsucht wurde, können auch einzelne Dateien analog zu Belkasoft mit integriertem Viewer betrachtet und manuell untersucht werden.

Axiom präsentiert in seinem Resource Center [15] den SQLite-Viewer, welcher verschiedene Funktionen wie Filtern der Spalten, SQL-Abfragen, wählbare Datentypen für Spalten und das Umwandeln von BLOB-Daten zulässt. Der Support von WALs wird nicht explizit erwähnt.

Wir konnten die Software über eine Testlizenz der Hochschule Wismar für diese Thesis einsetzen. Wir nutzen zu unserer Analyse die Version 8.1.0.40287

#### **4.2.3 Autopsy / Sleuthkit**

Die frei erhältliche Software Autopsy stellt eine grafische Benutzeroberfläche für das von Brian Carrier entwickelte Sleuth Kit zu Verfügung. Es handelt sich um ein forensisches Tool, das von der Strafverfolgung, Militär und Firmen genutzt wird, um Postmortem-Analysen zu erstellen und Daten wiederherzustellen [30].

Ursprünglich wurde es als grafische Benutzeroberfläche für das „The Coroner’s

Toolkit“ (TCT) entwickelt, auf welchem Sleuth Kit basiert.

Autopsy wird seit 2001 stetig weiterentwickelt, um mit den Fortschritten des Sleuth Kit mitzuhalten.

In Autopsy selbst ist eine Reihe von mitgelieferten Modulen enthalten, welche die Festplatten- oder Smartphone-Abbilder auf diverse Artefakte untersuchen kann. Darunter zählen unter anderem Telefonnummer, IP-Adressen, Bilder, E-Mailadressen, Geolokationsdaten und besuchte Webseiten.

Eine Erweiterung der angebotenen Module in Autopsy ist möglich. Diese können entweder von Dritten bezogen oder selbst geschrieben werden.

In unserem Fall gibt es im Github von Mark McKinnon zwei Python-Plugins (Parse\_SQLite\_Databases, Parse\_SQLite\_Del\_Records) für Autopsy. Diese wurden in unserer Installation hinzugefügt.

Weiterhin ist die Möglichkeit gegeben sich Dateien direkt in der Vorschau anzeigen zu lassen, um diese nach der automatisierten Analyse manuell zu untersuchen. Hierzu sind beispielsweise eingebaute Viewer für Registry-Hive oder SQLite-Datenbanken enthalten.

Autopsy steht für alle größeren Betriebssysteme (Windows, MacOS, Linux) als Java-Software zur Verfügung.

Wir nutzen zu unserer Analyse die Autopsy-Version 4.21 in Verbindung mit der Version 4.12.1 des Sleuthkits.

#### **4.2.4 FQLite**

FQLite ist eine in Java geschriebene Software zum Parsen von SQLite-Datenbanken. Die von Dirk Pawlaszczyk geschriebene und von der Hochschule Mittweida frei erhältlich angebotene Software kann Datensätze finden und wiederherstellen.

Die gefundenen Datensätze können per CSV-Export oder per Zeilenkopie aus der grafischen Benutzeroberfläche extrahiert werden.

FQLite ist kompatibel zu Rollback-Journals und WAL-Dateien. Ebenso werden Freelist-Seiten ausgelesen und es existiert ein eingebauter Hex-Viewer für eine tiefgehende Analyse beziehungsweise eine Überprüfung der gefundenen Werte.

Laut Webseite findet die Software im standardisierten Forensikkorpus 100% der Artefakte.

Des Weiteren zeigt die Software den Verlauf der Checkpoints einer WAL-Datei.

Wir nutzen zu unserer Analyse die Version 2.61.

## **4.3 Pythonskripts**

### **4.3.1 Bring2lite**

Bring2lite ist ein Tool, welches von Christian Meng und Harald Baier auf Digital Forensic Research Conference 2019 als Ergebnis ihrer Arbeit auf dem Gebiet der forensischen Analyse von SQLite-Datenbanken vorgestellt wurde. Es soll die forensische Analyse, welche bis dato meist über die Werkzeuge „dd“, „file“ oder „strings“ durchgeführt wurden, erleichtern.

Über einen Algorithmus durchwandert das Skript SQLite-Datenbanken, WAL oder Rollback-Journals. Die Ausgabe erfolgt als Datei im Format CSV. Im Falle eines WALs werden die einzelnen darin befindlichen Seiten durchsucht und valide Einträge in eine Datei pro Seite abgelegt. Es werden keine SALT-, Checksummen- oder Offset-Werte ausgelesen.

### **4.3.2 WAL Crawler**

Wal\_crawler.py ist eine Übungsdatei zu dem Buch „Learning Python for forensics – Second Edition“. Die Datei steht unter der MIT Lizenz und ist frei verfügbar. Das Skript durchsucht WAL-Dateien und gibt deren Inhalt in für Menschen lesbarer Form als CSV-Datei aus. Als Zusatzinformation werden Salt-, Frame- und Zellen-Offset abgelegt.

### **4.3.3 Wal2sqlite**

wal2sqlite ist ein im Rahmen der Thesis geschriebenes Skript, um die Ausgabe von bring2lite in die ursprüngliche Datenbank einzufügen. Somit kann in der Datenbank mittels SQL-Abfrage versucht werden weitere Infos zu erhalten, Zusammenhänge herzustellen oder diese besser zu erkennen.

Grundsätzlich liest das Skript dabei die einzelnen von bring2lite erstellten CSV-

Dateien ein. Wir erfassen von jeder Datei die erste Zeile und zählen anhand der auftretenden Kommas die wahrscheinlichen Spalten des Datensatzes.

Nun vergleichen wir die Anzahl der ermittelten Spalten mit den Spalten der einzelnen Tabellen der Datenbank. So erhalten wir eine einfache Zuordnung von Datei zu Tabelle.

Nachdem die einzelnen Dateien Tabellen in der Datenbank zugeordnet wurden, erstellt das Skript in der Datenbank eine Kopie der entsprechenden Tabelle und fügt dann die Zeilen und Werte ein. Damit die Herkunft der Zeilen nachvollzogen werden kann, wird in der kopierten Tabelle eine zusätzliche Spalte „log\_filename“ eingefügt. Dort ist der Name der Quelldatei abgelegt.

Per SQL-Abfrage sollte es dann möglich sein, Inhalte aus Original- und Kopietabelle abzufragen, um weitere forensische Hinweise zu erhalten.

Die Verwendung des Skripts ist einfach: Aufrufen des Skripts über dessen Dateiname gefolgt von der SQLite-Datenbank, in welche die Daten importiert werden sollen. Danach wird der Pfad angegeben, in welchem die Dateien aus bring2lite abliegen.

Der Quellcode ist im Anhang (9.7) aufgeführt.

#### **4.3.4 Freeblock\_seeking**

Da in den aufgeführten Softwarepaketen nicht eindeutig identifiziert werden kann, ob Freeblocks innerhalb einer SQLite-Datei behandelt werden, wurde ein Skript zum Auslesen der Freeblocks auf Basis DeGrazias [31] erstellt. Das Skript von DeGrazia hat bei uns nicht funktioniert, es wurden einige Fehler bei der Anwendung ausgeworfen.

Somit haben wir unser Skript auf dieser Basis entwickelt. Es liest die Freeblocks auf B-tree-Seiten in SQLite-Dateien und Write-Ahead-Logs aus und exportiert den Inhalt mit den nötigen Informationen in eine Textdatei.

Bei den Informationen handelt es sich um die Startadresse, Länge und die Endadresse des Freeblocks.

Die Verwendung des Skripts ist einfach: Aufrufen des Skripts über dessen Dateiname gefolgt von der SQLite-Datenbank oder dem Write Ahead Log und

abschließend der Name der Textdatei, in welche die extrahierten Daten geschrieben werden sollen.

Der Quellcode ist im Anhang (9.6) aufgeführt.

## **4.4 Weitere Software**

### **4.4.1 Visual Studio Code**

Visual Studio Code ist ein von Microsoft kostenlos und unter der MIT-Lizenz stehender angebotener Quelltext-Editor welcher auf allen größeren Betriebssystemen (Windows, MacOS, Linux) verfügbar ist.

Die Software erschien erstmalig im Jahr 2015 und wird bis heute aktiv weiterentwickelt.

Es werden eine Reihe von Skript-, Programmier- und Auszeichnungssprachen (z.B. C++, CSS, Lua, JavaScript, Python) unterstützt.

Durch ein Plug-In-System können u.a. zusätzliche Sprachen eingebaut oder Funktionen wie Intellisense, Debugging, Syntaxhervorhebung hinzugefügt werden.

Wir haben es zur Unterstützung der Erstellung unsere Pythonskripte „wal2sqlite“ „freeblock\_seeking“ genutzt. Es lag zum Zeitpunkt der Programmierung die Version 1.90.0 vor.

### **4.4.2 DB Browser for SQLite**

Auch der DB Browser for SQLite ist ein frei verfügbares Open-Source-Werkzeug zum Anzeigen von Inhalten einer SQLite-Datenbank.

Die Software wird seit 2003 entwickelt und ist aktuell in der Version 3.12.2, welche in der Thesis verwendet wird, verfügbar.

Das Tool umfasst einige Optionen zum Analysieren der SQLite-Datenbank. Auch sind das Erstellen und Ausführen von SQL-Abfragen möglich. Ebenso werden detailliert die Tabellen inklusive der Datentypen angezeigt.

Für die Software sind zwei Punkte festzuhalten:

Es handelt sich um eine dedizierte Software, um SQLite-Datenbanken zu durchsuchen und manipulieren. Sei es das Hinzufügen oder Entfernen von Datensätzen oder das Aktivieren / Deaktivieren von Pragma-Optionen.

„DB Browser for SQLite“ kann keine WALs interpretieren und deshalb ist beim Öffnen einer Datenbank mit vorhandener temporärer Datei darauf zu achten, dass diese umbenannt wird. Passiert dies nicht wird beim ordnungsgemäßen Schließen der Datenbank die temporäre Datei aufgelöst und der Inhalt ist nur schwer rekonstruierbar.

Somit weisen wir darauf hin, dass es sich bei dem genannten Tool nicht um eine für den IT-forensischen Kontext einzusetzende Software handelt, für die in dieser Thesis behandelten Experimente dennoch ausreichend ist.



## 5 Ergebnisse Szenario Firefox

Bei unserem ersten Szenario analysieren wir den Browserverlauf von Firefox unter Windows 10 in einer virtuellen Maschine. Explizit konzentrieren wir uns auf die Datei `places.sqlite` und `favicons.sqlite`. Beide Datenbanken sind im Userverzeichnis des angemeldeten Nutzers abgelegt.

`Places.sqlite` bildet den Browserverlauf, Downloads und Lesezeichen ab. Des Weiteren werden Telemetriedaten wie Scrollabstand und Verweildauer auf einzelnen Webseiten abgebildet.

Weiterhin beziehen wir die Datei `favicons.sqlite` ein. Hier werden kleine Vorschaubilder der bereits aufgerufenen Seiten abgelegt, um diese in der Adressleiste anzuzeigen.

Mit dem Browser wurden zuvor diverse Seiten aufgerufen, der Browserverlauf gelöscht, um danach neue Seiten aufzurufen. Der Ablauf ist in Anhang A beschrieben.

Damit das Write-Ahead-Log in die forensische Untersuchung einbezogen werden kann, darf Firefox nicht auf herkömmlichen Weg (Datei -> Beenden; über das X am Fensterrand) geschlossen werden. Dies hat die Folge, dass die Daten des WALs in die produktive Datenbank zurückgeschrieben werden. Die WAL-Datei ist noch im Dateisystem sichtbar, aber hat die Größe von 0 Kilobyte.

Um das Write-Ahead-Log zu erhalten muss Firefox über den Taskmanager oder mit dem Befehl „`taskkill /F /IM firefox`“ per Kommandozeile zum Schließen gezwungen werden. In diesem Fall schreibt die Applikation den Inhalt des Write-Ahead-Logs nicht in die produktive Datenbank zurück.

Die zu Firefox und dem aktuell angemeldeten Benutzer gehörenden SQLite-Datenbanken liegen unter Windows in folgendem Pfad ab:

```
C:\Users\%Benutzer%\AppData\Roaming\Mozilla\Firefox\Profiles\%Zufallsstring%.default-release\
```

## 5.1 Belkasoft (Szenario Firefox)

In Belkasoft können wir nach Eröffnung eines Falles eine Datenquellen oder ein bereits bestehendes Abbild angeben. Wir wählen den Punkt Image und verweisen auf die VDI-Dateien der virtuellen Windows 10-Maschinen.

Darauffolgend geben wir an nach welchen Artefakten Belkasoft das Image durchsuchen soll. Der Einfachheit halber wurden alle Artefakte für ein Windows-Betriebssystem aktiviert.

Nachdem die Durchsuchung der Festplattenabbilder durch Belkasoft abgeschlossen ist, navigieren wir in den Reiter „Artifacts“ und dort in die Gesamtübersicht „Overview“. Unter dem Punkt „Browsers -> URLs“ setzen wir in der Spalte „Type“ den Filter für Firefox, um das Ergebnis einzugrenzen. Die 32 angezeigten Artefakte beziehen sich auf beide Festplattenabbilder.

Wählt man die einzelnen Artefakte aus, bekommen wir einen ersten Überblick der Informationen des Eintrags. Bei diesen Informationen handelt es sich um die Lokation des Eintrags, also aus welcher Datenbank diese Information extrahiert wurde, die Länge des Eintrags in der Datenbank und deren Offset in Bytes innerhalb der Datei.

Mit dem Eintrag der Lokation können wir beim ersten Durchsehen der gefundenen Informationen feststellen, dass ein überwiegender Teil aus dem WAL der Datenbank stammt.

Des Weiteren ist über die Spalte „is deleted“ von Belkasoft aufgeführt, ob der Eintrag für eine Löschung vorgesehen ist oder nicht. Dies wird so auf der Seite von Belkasoft zu SQLite Forensics postuliert [14]. Die Spalte zeigt den Wert „yes“ bei der Webseite „ebay.de“ an, diese ist allerdings in der Datenbank und im WAL auffindbar. Daher ist für uns unklar, wie Belkasoft zu diesem Schluss kommt.

Wir haben versucht die Information nachzuvollziehen. Dazu haben wir von den drei Einträgen, die als gelöscht markiert sind, zwei analysiert. Beide Einträge sind im WAL der Verlaufsdatenbank places.sqlite zu finden. Beide Einträge liegen am Ende eines Freeblocks, das heißt der Freeblock hört genau vor dem von Belkasoft markierten Eintrag auf. Der Eintrag steht dort auch nicht allein. Es folgen weitere Zeilen der entsprechenden Tabelle, die auf dieser B-tree-Seite

abgebildet sind.

Sofern wir weitere Informationen aus der Datenbank ermitteln möchten, wechseln wir auf die „File System“-Ansicht. Dort können wir in beiden Festplattenabbildern zur Datei places.sqlite navigieren und starten den eingebauten SQLite-Viewer.

In der darauffolgenden Ansicht bekommen wir die Eigenschaften der Datenbank und deren Tabellen zu sehen.

Open file: image:\2\vol\_122683392\Users\blackhand\AppData\Roaming\Mozilla\Firefox\Profiles\rbcjcg7t.default-release\places.sqlite

Properties	Properties
<ul style="list-style-type: none"> <li>▼ Tables</li> <li>    moz_anno_attributes (0)</li> <li>    moz_annos (0)</li> <li>    moz_bookmarks (12)</li> <li>    moz_bookmarks_deleted (0)</li> <li>    moz_historyvisits (82)</li> <li>    moz_historyvisits_extra (0)</li> </ul>	<ul style="list-style-type: none"> <li>Page size: 32768</li> <li>Number of pages: 160</li> <li>Number of freelist pages: 0</li> <li>Journal type: Wal</li> <li>Encoding: UTF-8</li> <li>MD5: 1F84B55B0B3E756499AEB6959FA55C5D</li> <li>SHA1: FAACDA77BB5661F27BED45BD54F4ED56274DA891</li> </ul>

**Abbildung 5-1 Eigenschaften der Datenbank places.sqlite**

Unter den Eigenschaften sind erste, für weitere forensische Untersuchungen, relevante Informationen wie Seitengröße und Anzahl sowie der Journal-Modus, Encoding und Anzahl der Freelists-Seiten aufgeführt.

Was nicht aufgeführt wird, ist die Angabe der Gesamtstruktur der einzelnen Tabellen. Es ist somit nicht nachzuvollziehen, ob unter den einzelnen Tabellen Verbindungen in Form von Fremdschlüsseln existieren. Es ist nicht möglich die „sqlite\_master“-Tabelle auszulesen, da die Funktion für SQL-Abfrage nicht gegeben ist.

Über die Auswahl der einzelnen Tabellen wird mit Hilfe des Reiters „Structure“ die einzelnen Datentypen der Spalten inklusive des Primärschlüssels nicht aber Fremdschlüssel angezeigt.

Um die zuletzt aufgerufenen Webseiten nachzuvollziehen, analysieren wir die Tabelle „moz\_places“. Ein Großteil der Einträge stammt aus dem Write-Ahead-Log. Wir schauen uns den Eintrag für das Aufrufen der Seite spiegel.de an:

<input type="checkbox"/>	↓ Row id	Record type	id [Primary ke	origin_id	url	last_visit_date
<input type="checkbox"/>	6		6	3	http://spiegel.de/	1718446461837000
<input type="checkbox"/>	6	Wal	6	3	http://spiegel.de/	1718446461837000
<input type="checkbox"/>	6	Wal	6	3	http://spiegel.de/	1718446461837000

**Abbildung 5-2 Ausschnitt der Tabelle moz\_places**

Es ist erkennbar, dass Firefox den identischen Eintrag mehrfach in das WAL schreibt. Für jede aufgerufene Seite sind es im ersten Festplattenabbild zehn Einträge und im zweiten nur acht. Ohne die Angabe des Salts aus dem WAL kann nicht eindeutig festgestellt werden, wann Änderungen an den Einträgen durchgeführt wurden. Auch ist ein Feststellen von Änderungen über die Information der Spalte „last\_visit\_date“ nicht möglich. Diese hat für jeden Eintrag in der Datenbank und im WAL den gleichen Wert.

Aus den vorliegenden Informationen ist zu schließen, dass der Großteil der Einträge aus der Tabelle „moz\_places“ auf einer Seite der Datenbank festgehalten wird. Diese Annahme kann bei Prüfung weiterer in der Tabelle stehenden Einträge geschlussfolgert werden, da immer die gleiche Anzahl an WAL-Einträgen vorhanden ist. Ein weiteres Indiz dafür ist die von Firefox groß gewählte Seitengröße von 32768 Bytes. Einträge in der Tabelle besitzen eine Größe von ca. 100 – 500 Bytes.

Die „last\_visit\_date“-Spalte kann in dieser Ansicht von Belkasoft nicht in ein für Menschen leichter lesbares Format umgewandelt werden. Ein Anpassen der Anzeige für die Spalte ist zwar möglich, allerdings speichert Firefox die Besuchszeit im Format Linux Epoch in Millisekunden. Diese Option ist nicht verfügbar und eine benutzerdefinierte Einstellung für die Formatierung der Spalte nicht möglich.

Hier muss, sofern benötigt, mit einem Workaround gearbeitet werden:

Die einzelnen Einträge können aus der Ansicht unter anderem in das Excel-Format xlsx exportiert werden. Über eine zusätzliche Spalte kann die Angabe mit Hilfe der folgenden Formel in ein für Menschen lesbares Datum umgewandelt werden:

`=((Zellwert/1000000)/86400)+(DATWERT("1-1-1970"))`

id [Primary key]	url	last_visit_date	Benutzerdefiniert
25	http://hardwareluxx.de/	1718531696873000	16.6.24 9:54
29	https://computerbase.de/	1718531747966000	16.6.24 9:55
30	https://www.computerbase.de/	1718531748004000	16.6.24 9:55
32	https://www.computerbase.de/2024-06/	1718531848773000	16.6.24 9:57
34	https://ebay.de/	1718536757623000	16.6.24 11:19

**Abbildung 5-3 Umrechnung des Datumformats**

Mit Hilfe des zweiten Festplattenabbildes wurde festgestellt, dass Firefox die Eintragung in die Datenbank restriktiv handhabt. Alte Einträge werden aus der produktiven Datenbank und dem WAL gelöscht. Allerdings haben wir zwei Indikatoren in unserem Experiment festgestellt anhand derer Lücken im Verlauf vermutet werden können. Diese Annahmen werden auch im Buch von Sanderson [22] aufgeführt.

Zum einen kann in der SQLite-Viewer-Ansicht die Spalte „Row id“ eingeblendet werden. Dort sind im zweiten Abbild Zahlenlücken bei Tabellen festzustellen. Als Beispiel ist die Tabelle „moz\_places“ genannt. Hier können im zweiten Abbild mittels „Row id“ fehlende Einträge zwischen 5 und 25 identifiziert werden. Um dies zu verifizieren, schauen wir uns die Tabelle in Abbild Eins an. Es findet sich eine aufsteigende ununterbrochene Zahlenreihe von 1 bis 24. Die Überschneidung der Zeilen Eins bis Vier ergibt sich aus der aufgerufenen Webseite, dabei handelt es sich um eine Mozilla-eigene Seite, die standardmäßig beim ersten Start von Firefox nach der Installation aufgerufen wird. Zum anderen kann über die Spalte „origin\_id“ ein Fehlen von Datensätzen nachgewiesen werden.

In der Tabelle „moz\_origins“ werden alle aufgerufenen Seiten katalogisiert. Diese erhalten eine ID, welche als Primary Key geführt wird. Sofern wir uns die Datensätze der Tabelle „moz\_places“, dort die Spalte „origin\_id“ ansehen können wir eine Verbindung zum Primary Key in „moz\_origins“ herstellen.

Im ersten Festplattenabbild ist die höchste „id“ mit der Nummer 9 versehen. Im zweiten Festplattenabbild fängt die Zählung der „id“ mit Nummer 10 an. Somit wäre dies ein Indikator für eine Löschung des Browserverlaufs.

Würde es sich um eine größere Datenbank handeln könnte man hier unter Umständen eine Zeitspanne identifizieren, welche im Verlauf fehlt.

Für weitere Anhaltspunkte auf gelöschte Daten aus dem Verlauf ist eine Analyse der Datei „favicons.sqlite“ möglich. Bei unseren Beobachtungen haben wir festgestellt, dass Firefox Logos der aufgerufenen Webseiten ablegt und das Ablegen der Informationen nicht so restriktiv handhabt wie in der Verlaufsdatenbank.



Diese erscheinen beim Eingeben einer Adresse in der Vorschlagsliste und Adressleiste. Die Icons liegen in der Datenbank als BLOB vor.

Eine Analyse der Datenbank aus Festplattenabbild Zwei zeigt die Logos für die Seite reddit.com und youtube.com. Laut Belkasoft kommen die Einträge aus dem WAL der Datenbank.

Diese beiden Seiten tauchen in der Datei places.sqlite nicht auf. Daher ist davon auszugehen, dass die Seiten aus dem Verlauf gelöscht wurden.

392\Users\blackhand\AppData\Local\Roaming\Mozilla\Firefox\Profiles\rbcjg7.default-release\favicons.sqlite

Items: 95 Number of journal records: 86

Record type	id (Primary key)	icon_url	fixed_icon_url_hash	width	root	color	expire_ms	data
	12	https://www.redditstatic.com/shreddit/assets/favicon/192x192.png	2921847948	192	0		1719051438924	
	13	https://www.reddit.com/favicon.ico	2282350422	32	1		1719069372846	

**Abbildung 5-4 Artefakte in favicons.sqlite**

Über die Spalte „expire\_ms“ ist es möglich eine Rückdatierung des letzten Abrufs für das VorschauBild zu unternehmen. Sofern wir die Aufrufzeit der Seite reddit.com aus dem ersten Abbild prüfen und den Wert der angesprochenen Spalte gegenüberstellen, ergibt sich eine Differenz von sieben Tagen. Das heißt sofern wir ein solches Fragment finden, können wir den letzten Aufruf der Seite ermitteln. Diese Annahme müsste insofern verifiziert werden, als dass in Firefox keine andere Programmroutine mit einem regelmäßig im Hintergrund stattfindenden Download entgegensteht.

Schließlich bietet Belkasoft für jede Datenbank noch den Punkt „Unallocated space“ an. Es könnte sich hier entweder um den Inhalt von Freelists oder Freeblocks der Datenbank handeln. Sofern wir die Erklärung von Belkasoft [14] richtig deuten, finden sich hier Artefakte wieder, die sich im nicht zugewiesenen

Speicherplatz einer Seite zwischen Ende des Seitenheaders und dem Anfang der Zellen Daten am Ende der Seite befinden. Im vorliegenden Fall sind keine verwertbaren Informationen zu finden. Nur Namen von Webseiten, die bereits in der Datenbank sichtbar sind:

Items: 12

[Carved data](#) [Raw data](#)

<input type="checkbox"/>	URL	Name	Creation time (local)	Offset (bytes)	Len
<input checked="" type="checkbox"/>	https://www.hardwareluxx.de			97964	84
<input type="checkbox"/>	https://hardwareluxx.de			97997	51
<input type="checkbox"/>	http://hardwareluxx.de			98026	22
<input type="checkbox"/>	https://ebay.dehttp://ebay.de 3			98117	187
<input type="checkbox"/>	http://ebay.de 3			98138	166

**Abbildung 5-5 Unallocated Space der Datenbank places.sqlite**

Auch hier haben wir zumindest die ersten drei Einträge genauer untersucht. Die Lokation der Einträge ist die Verlaufsdatenbank „places.sqlite“. Wir können verifizieren, dass sich die Einträge auf einer Index B-tree-Seite befinden. In diesem Fall sind die Einträge zwischen dem Seitenheader und Start des ersten und einzigen Freeblocks der entsprechenden Seite.

## 5.2 Axiom (Szenario Firefox)

Wie zuvor bei Belkasoft können wir in Magnet Axiom einen Fall eröffnen und daraufhin als Datenquellen die beiden erstellten Festplattenabbilder angeben.

Die Einstellungen zur Suche nach Artefakten haben wir, bis auf eine Einstellung, bei den Standard-Einstellungen belassen. Das Durchsuchen von Archiven wurde aktiviert. Somit werden unter anderem SQLite-Datenbanken eingeschlossen.

Sobald die beiden Festplattenabbilder untersucht worden sind, können wir uns in der „Artifacts“-Übersicht unter der Kategorie „WEB RELATED“ die gefundenen „Firefox Web Visits“ ansehen. Die Anzahl der gefundenen Besuche für beide Abbilder beläuft sich auf 32. Damit haben wir die gleiche Anzahl zu Belkasoft.

Auch hier bekommen wir in der Übersicht erste forensische Daten:

URL	Location	Date Visited Date/Time	Typed	Transition Type
http://spiegel.de/	Table: moz_places(id: 6), Table: moz_historyvisits(id: 1)	15.06.2024 10:14:21.837	Yes	TRANSITION_TYPED
https://www.spiegel.de/	Table: moz_places(id: 8), Table: moz_historyvisits(id: 3)	15.06.2024 10:14:22.117	No	TRANSITION_REDIRECT_PERMANENT
https://spiegel.de/	Table: moz_places(id: 7), Table: moz_historyvisits(id: 2)	15.06.2024 10:14:21.961	No	TRANSITION_REDIRECT_PERMANENT

Abbildung 5-6 Anzeige Web-Related unter Axiom

Wie bei Belkasoft sehen wir uns den Aufruf von Spiegel.de an. Zum einen sehen wir aus welcher Tabelle der Eintrag stammt, zum anderen zeigt uns die Referenz auf „moz\_historyvisits“ eine Verbindung zwischen den beiden Tabellen. Unter Umständen kann hier nachvollzogen werden, ob die Datenbank manipuliert worden ist sofern der Eintrag für die Tabelle „moz\_historyvisits“ fehlt.

In der Übersicht der Artefakte sind unter „Firefox FavIcons“ die Inhalte der Datenbank favicons.sqlite einsehbar. Axiom zeigt uns hier keine verwaisten Einträge an. Dies lässt darauf schließen, dass Write-Ahead-Logs vom SQLite-Viewer nicht unterstützt werden.

Auch bei Axiom versuchen wir tiefere Informationen direkt aus der SQLite-Datenbank zu sichten, indem wir die Datei über die „File system“-Ansicht aufrufen.

Der Vorteil des SQLite-Viewers in Axiom ist die Möglichkeit der Abfrage der Datenbank via SQL.

places.sqlite

```
select * from sqlite_master where type = 'table'
```

SQLITE

Select ta

CLEAR EXECUTE

FIND BUILD QUERY EXPORT

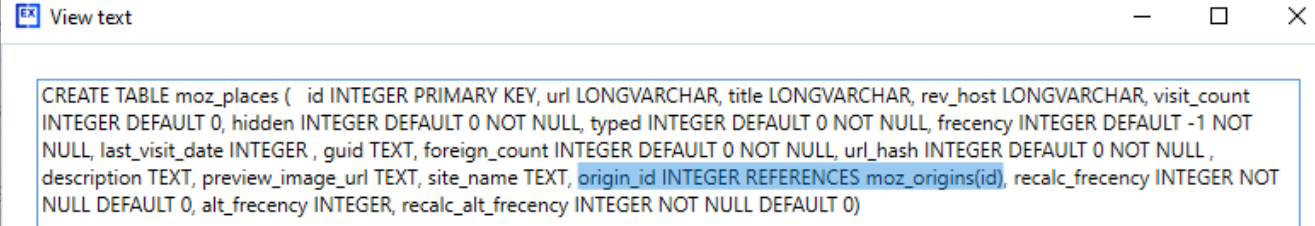
#	type	name	tbl_name	rootpage	sql
1	table	moz_origins	moz_origins	2	CREATE TABLE moz_origins ( id INTEGER PRIMARY KEY,...
2	table	moz_places	moz_places	4	CREATE TABLE moz_places ( id INTEGER PRIMARY KEY...
3	table	moz_places_extra	moz_places_extra	5	CREATE TABLE moz_places_extra ( place_id INTEGER ...
4	table	moz_historyvisits	moz_historyvisits	14	CREATE TABLE moz_historyvisits ( id INTEGER PRIMA...

Abbildung 5-7 SQL-Abfrage der sqlite\_master in Axiom

Mit der obenstehenden Abfrage „SELECT \* FROM sqlite\_master WHERE type = ‚table““ lassen wir uns alle Tabellen der Datenbank anzeigen. Allerdings wird der SQL-Befehl in der Spalte „sql“ nicht komplett angezeigt, auch bei einem Export in „CSV“ oder „XLSX“ nicht. Wir können uns den gesamten Text mit einem



Rechtsklick auf die Zelle anzeigen lassen:

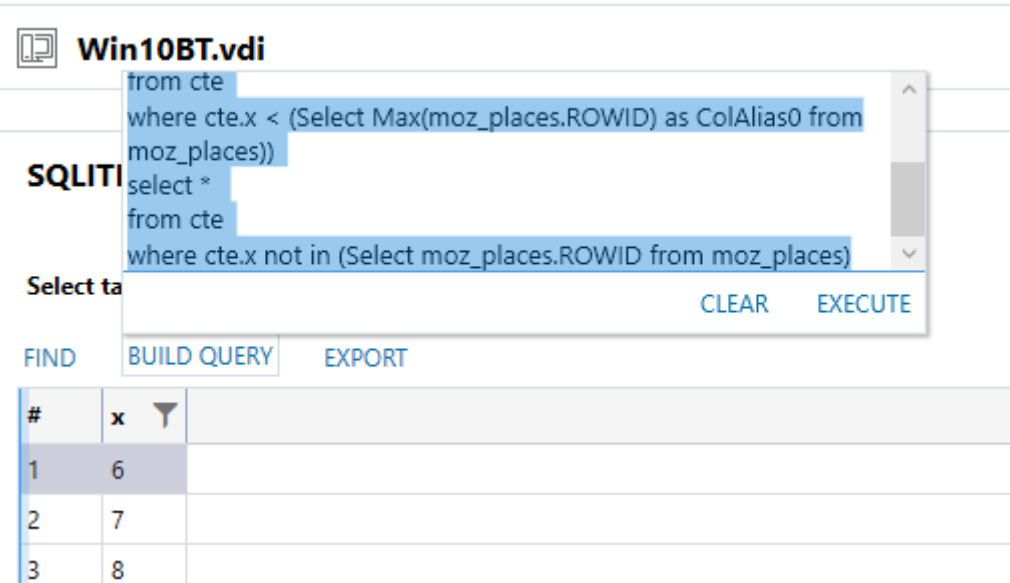


```
CREATE TABLE moz_places ( id INTEGER PRIMARY KEY, url LONGVARCHAR, title LONGVARCHAR, rev_host LONGVARCHAR, visit_count INTEGER DEFAULT 0, hidden INTEGER DEFAULT 0 NOT NULL, typed INTEGER DEFAULT 0 NOT NULL, frecency INTEGER DEFAULT -1 NOT NULL, last_visit_date INTEGER, guid TEXT, foreign_count INTEGER DEFAULT 0 NOT NULL, url_hash INTEGER DEFAULT 0 NOT NULL, description TEXT, preview_image_url TEXT, site_name TEXT, origin_id INTEGER REFERENCES moz_origins(id), recalc_frecency INTEGER NOT NULL DEFAULT 0, alt_frecency INTEGER, recalc_alt_frecency INTEGER NOT NULL DEFAULT 0)
```

Abbildung 5-8 Create Table Statement der Tabelle moz\_places

Somit kann nachvollzogen werden, dass die Spalte „origin\_id“ der Tabelle „moz\_places“ auf die Spalte „id“ der Tabelle „moz\_origins“ referenziert wird. Auch können gewählte Einstellungen für die Datenbank im Falle einer Löschung eines Datensatzes („CASCADE ON DELETE“) nachvollzogen werden. Somit wäre es über SQL-Abfragen möglich manuelle Manipulationen an der Datenbank sichtbar zu machen.

Weiterhin können wir mit einer angepassten Version von Abfrage 1 [22, p. 268] komfortabel Lücken in der „id“- oder ROWID-Spalte nachvollziehen:



```
from cte
where cte.x < (Select Max(moz_places.ROWID) as ColAlias0 from
moz_places))
select *
from cte
where cte.x not in (Select moz_places.ROWID from moz_places)
```

#	x
1	6
2	7
3	8

Abbildung 5-9 Abfrage zur Feststellung von Lücken bei ROWID der Tabelle moz\_places

Als Ergebnis dieser Abfrage erhalten wir die „id“ oder ROWID der fehlenden Zeilen.

Ein Nachteil des eingebauten SQLite-Viewers ist die nicht vorhandene Möglichkeit der Abfrage von PRAGMAs innerhalb der Datenbank. Eine Abfrage nach der Einstellung zu secure\_delete wird mit einem „syntax error“ quittiert.

Da Axiom den WAL automatisch nicht in die Artefakte mit einbezieht ist es, zumindest für den Browserverlauf in Firefox, schwierig gelöschte Daten wiederherzustellen.

Ein Versuch zur Analyse des WALs ist das manuelle Auswählen der Datei und die Betrachtung mittels „Preview“-Fenster in der „File system“-Ansicht. Allerdings wird uns hier kein Inhalt angezeigt und somit ist die Datei auf diesem Weg nicht durchsuchbar.

Eine Stringsuche im angebotenen „Text and Hex“-Fenster ist möglich. Hier werden allerdings nur Strings bis 32 Zeichen, aber keine Regex-Ausdrücke unterstützt. Sofern man auf der Suche nach einem speziellen Inhalt ist, könnte dieser Weg gewählt werden.

### **5.3 Frei verfügbare Software (Szenario Firefox)**

Im Abschnitt der frei verfügbaren Software eröffnen sich zur forensischen Analyse vielfältige Möglichkeiten. Zum einen sind die Festplattenabbilder mit Autopsy durchsuchbar und zum anderen können wir die einzelnen Datenbanken mittels FQLite detailliert durchsuchen. Eine weitere Möglichkeit, um Nachteile von Autopsy, FQLite oder kommerzieller Software auszugleichen, liegt im Erstellen und Verwenden von Pythonskripten um die Analyse benutzerdefiniert anzupassen.

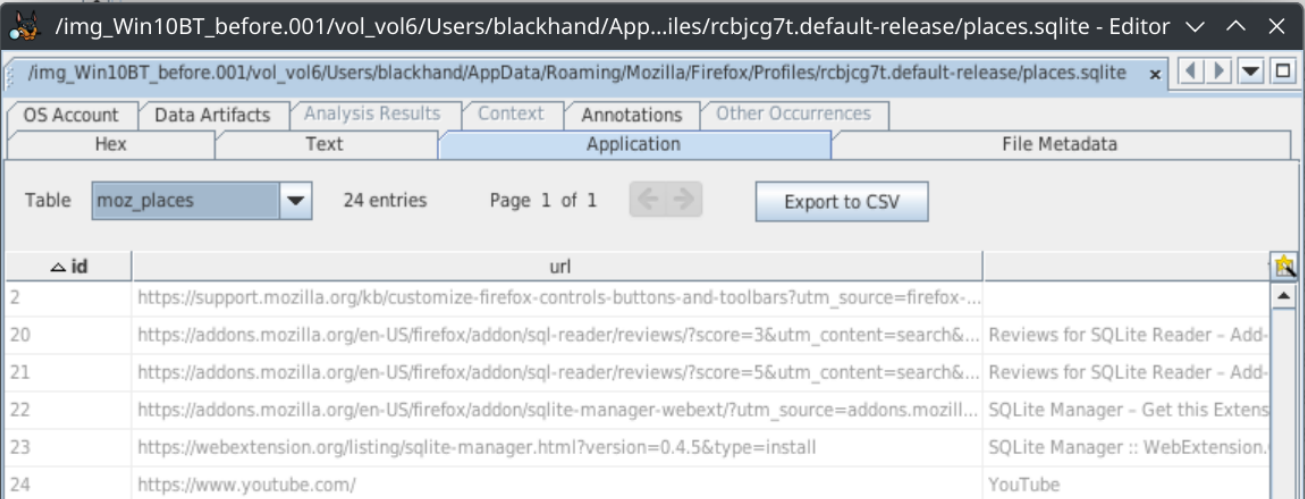
#### **5.3.1 Autopsy / Sleuthkit**

In Autopsy können, wie zuvor auch, unsere beiden Festplattenabbilder eingelesen und ausgewertet werden. Auch hier ist es möglich verschiedene Module zu aktivieren / deaktivieren, um bestimmte Artefakte zu finden. Um die Auswertung nicht zu zeitintensiv zu gestalten, wurde hier nur das Modul „Recent Activity“ ausgewählt. Bei anderen getesteten Konfigurationen konnten wir eine Auswertungsdauer von mehreren Stunden beobachten. Diese Analysezeiten für, in unserem Fall, drei SQLite-Datenbanken erachteten wir als nicht zielführend.

Das Modul „Recent Activity“ listet für uns die wichtigsten Artefakte auf. In der Artefakt-Sammlung unter dem Punkt „Web History“ sind alle gefundenen Webseiten gelistet. Ein erster Blick zeigt, dass zur Auswertung nur die Datei

„places.sqlite“ genutzt worden ist. Eine automatische Auswertung des WALs fand augenscheinlich nicht statt.

Weiterhin ist bei der Sichtung aufgefallen, dass Webseiten, welche in der Tabelle „moz\_places“ als besucht aufgeführt sind nicht in der Artefakt-Sammlung wiederzufinden sind. Explizit geht es hier um die Webseite „ebay.de“ im zweiten Festplattenabbild und um „youtube.com“ im ersten Festplattenabbild. Dieses Fehlen kann über den eingebauten SQLite-Viewer, welcher keine SQL-Abfragen ermöglicht, in Autopsy manuell recherchiert werden:



id	url	
2	https://support.mozilla.org/kb/customize-firefox-controls-buttons-and-toolbars?utm_source=firefox-...	
20	https://addons.mozilla.org/en-US/firefox/addon/sql-reader/reviews/?score=3&utm_content=search&...	Reviews for SQLite Reader - Add...
21	https://addons.mozilla.org/en-US/firefox/addon/sql-reader/reviews/?score=5&utm_content=search&...	Reviews for SQLite Reader - Add...
22	https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager-webext/?utm_source=addons.mozill...	SQLite Manager - Get this Extens...
23	https://webextension.org/listing/sqlite-manager.html?version=0.4.5&type=install	SQLite Manager :: WebExtension.
24	https://www.youtube.com/	YouTube

**Abbildung 5-10 Tabelleninhalte der Datenbank places.sqlite in Autopsy**

Als nächstes prüfen wir, ob der WAL der Datei places.sqlite über Autopsy einsehbar ist. Dazu navigieren wir über die Dateiansicht in das Profil-Verzeichnis von Firefox. Nach einem Klick auf die Datei kann diese entweder in der Hex- oder Textansicht angesehen werden. Eine Möglichkeit der Suche innerhalb des angezeigten Textes, welcher je nach Dateigröße in bis zu 150 Einzelseiten aufgeteilt ist, ist nicht möglich

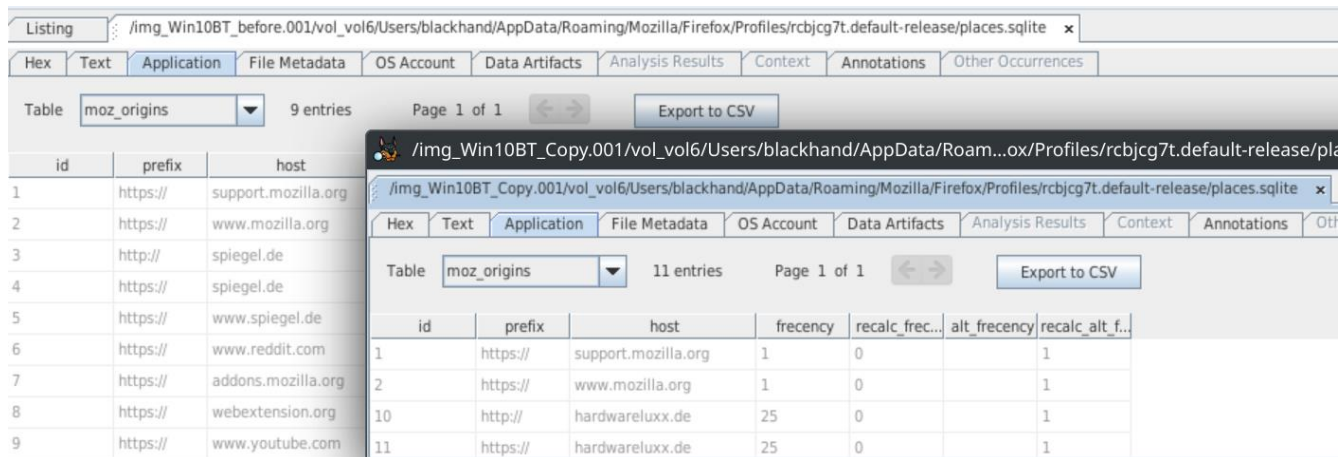
Somit ist es auch nicht, wie unter Belkasoft beschrieben, möglich mittels WAL der Datei „favicons.sqlite“ verwaiste Einträge zu Webseiten und deren Vorschaubilder zu extrahieren, um so Webseitenbesuche nachzuvollziehen.

Als nächstes wurde geprüft, ob die beiden eingebundenen Plugins für SQLite-Datenbanken weitere Informationen extrahieren können.

Uns war es nicht möglich mittels Plugins weitere Informationen aus den beiden Dateien zu extrahieren. Es ist auch nicht nachvollziehbar, ob die Plugins

irgendetwas durchlaufen haben. Es gibt keinerlei Logfile um Tätigkeiten nachzuprüfen.

Wir können unter Autopsy den in Axiom und Belkasoft beschriebenen Ansatz verfolgen und uns die „id“-Lücken in der Tabelle „moz\_origins“ von „places.sqlite“ anzeigen lassen:



**Abbildung 5-11 Manuelles Finden von fehlenden IDs in der Datenbank places.sqlite**

Hier ist anhand der der fehlenden id's im zweiten Festplattenabbild (Nummer drei bis neun) ersichtlich, dass eine Löschung im Verlauf stattgefunden hat.

### 5.3.2 FQLite

Nachdem wir die Analyse unter Autopsy beendet haben, wurden die Datenbanken „places.sqlite“ und „favicons.sqlite“ aus dem zweiten Festplattenabbild extrahiert und mit FQLite eingelesen.

Erste Analysen der „places.sqlite“ und deren WAL legen nahe, dass Firefox in dieser Datenbank die Pragma-Option „auto\_vacuum“ aktiviert hat da in beiden Dateien der Ordner „\_\_Freelist“ leer ist. Ein anderer Grund für das Nichtvorhandensein von Einträgen könnte die Anzahl der abgelegten Einträge sein. Die sind bei der Seitengröße zu gering.

Bei einer Stichprobe zum Vergleich verschiedener Einträge in den Tabellen wurde festgestellt, dass ein Großteil der abgelegten Informationen im WAL bereits in die produktive Datenbank übernommen worden ist:

No.	Offset	PLL	HL	id	place_id	referrer_place_id	created_at	updated_at	total_view_time	scrolling_time	scrolling_distance
1	1310629	29	12	1	27	null	1718531697679	1718531753978	32825	0	0
2	1310685	33	12	2	30	null	1718531748317	1718531849618	68978	4317	714

Abbildung 5-12 Ausschnitt von Einträgen aus der Datenbank mittels FQLite

No.	id	place_id	salt1	salt2	created_at	updated_at	total_view_time	scrolling_time	scrolling_distance
1	1	27	3497317630	625737469	1718531697679	1718531753978	32825	0	0
2	2	30	3497317630	625737469	1718531748317	1718531849618	68978	4317	714

Abbildung 5-13 Ausschnitt von Einträgen aus dem WAL mittels FQLite

Als Beispiel dient hier der Vergleich der Tabelle „moz\_places\_metadata“.

In der zweiten Datenbank „favicons.sqlite“ können wir feststellen, dass die Pragma-Option „auto\_vacuum“ wahrscheinlich nicht aktiv ist. Hier ist der Ordner „\_\_FREELIST“ des WALs befüllt. Dort kann auch der Fund aus Belkasoft verifiziert werden. Es kann dort der gelöschte Eintrag zum Vorschaubild für die Webseite „reddit.com“, inklusive der BLOB-Daten, gefunden werden:

No.	Offset	PLL	HL	co...	dbpage	w...	salt1	salt2	col1	col2
1	639034	16834	11	true	12	19	2335989613	3181781416	null	https://www.redditstatic.com/shreddit/assets/favicon/192x192.png

Abbildung 5-14 Gefundenes Artefakt der Webseite reddit.com mittels FQLite

### 5.3.3 Pythonskripte

Über die in 4.3 genannten Pythonskripte bring2lite und wal-crawler haben wir versucht die Einträge des WALs per SQL durchsuchbar zu machen, um diese mit den Daten der Produktiv-Datenbank in Verbindung zu setzen. Die Idee dahinter ist über SQL-Abfragen weitere Informationen zu erhalten.

Zur Realisierung des Vorhabens kam das von uns geschriebene Skript zum Einsatz. Allerdings sind wir in unserem Experiment auf Hindernisse für dieses Szenario gestoßen.

Einige Webseiten geben eine Titelangabe beim Aufruf mit. Diese wird in der Datenbank abgelegt und enthält, je nach Seite und Kreativität des Betreibers, ganze Sätze inklusive Kommas.

Somit erfolgt beim Einlesen der Zeilen ein Fehler, der in einer zu hohen Anzahl von Spalten resultiert. Damit kann der darauffolgende INSERT-Befehl von SQL nicht mehr durchgeführt werden:



Als Beispiel haben wir die Datei favicons.sqlite aus dem zweiten Festplattenabbild bearbeitet:

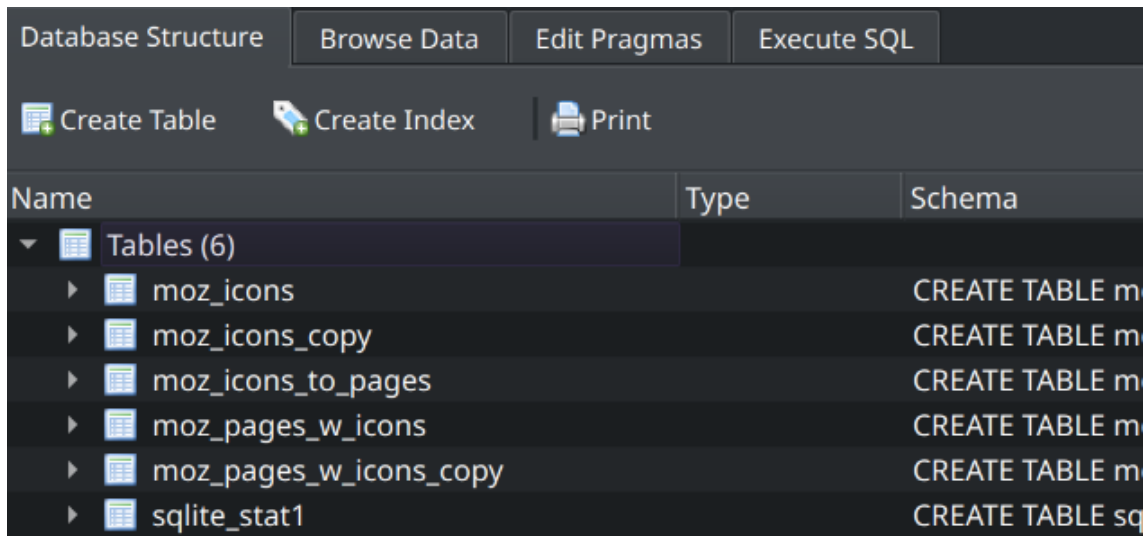


Abbildung 5-17 Überblick der Tabellen nachdem Einträge aus dem WAL hinzugefügt wurden

Nach dem Einfügen der Datensätze sind unter Datenbankstruktur die jeweiligen Kopien der Tabellen sichtbar.

Um nun verwertbare Hinweise zu sichten, arbeiten wir mit einer Abfrage (siehe 9.3) und stellen fest, welche URLs im WAL zu finden sind, die nicht in der produktiven Datenbank aufgeführt sind:

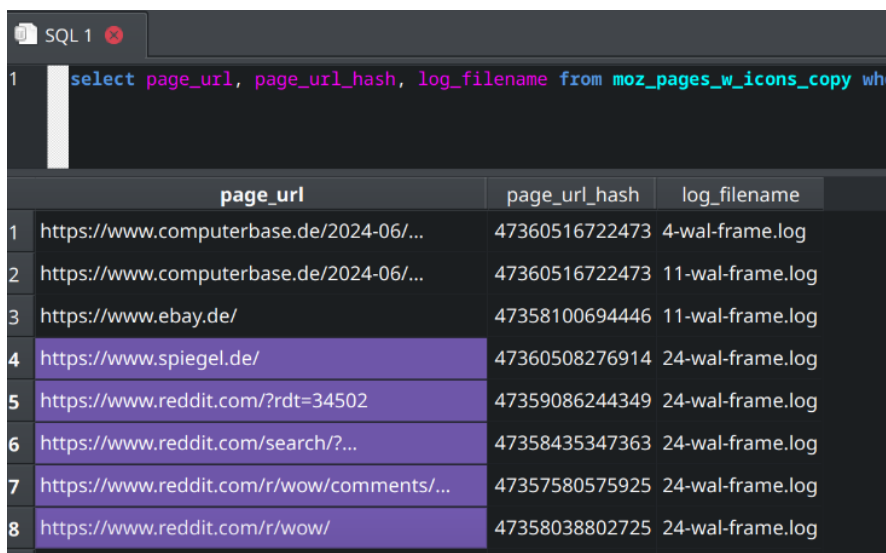


Abbildung 5-18 Einträge die nicht in der Produktivdatenbank zu finden sind

Für die Datenbank places.sqlite können wir über eine angepasste SQL-Abfrage (siehe 9.1) Lücken in der „Row id“ finden:



```

1 with recursive cte(x) as (Select (Select Min(moz_places.ROWID) as ColAlias0 from moz_places) AS ColAlias0
2 union all
3 select cte.x + 1 as colAlias1
4 from cte
5 where cte.x < (Select Max(moz_places.ROWID) as ColAlias0 from moz_places))
6 select *
7 from cte
8 where cte.x not in (Select moz_places.ROWID from moz_places)

```

x
6
7
8
9

Abbildung 5-19 Ausschnitt der fehlenden ROWIDs

Weiterhin können wir über die eingesetzte Software die forensisch relevanten Pragma-Optionen abfragen:

Tabelle 5-1 PRAGMA-Optionen der Datenbanken aus Firefox

	secure_delete	auto_vacuum
places.sqlite	1 (On / True)	0 (Off)
favicons.sqlite	1 (On / True)	2 (incremental)



## 6 Ergebnisse Szenario Memorix

Bei Szenario Nummer Zwei greifen wir auf eine Datenbank einer Android-App zurück. Bei der App handelt es sich um „Memorix Notizen + Checklisten“ (ID im Appstore: „panama.android.notes“). Die Datenbank stammt von einem mit Custom-Rom ausgestatteten Android-Smartphone der Marke Nokia 6.1. Trotz des Custom Rom war kein globaler Root-Zugriff möglich. Daher konnten wir weder über Belkasoft, Axiom noch per Android Debug Bridge ein entschlüsseltes Backup oder Image erstellen. Die Datenbank konnte aber über die Root-Funktion von ADB extrahiert werden. Die SQLite-Datei inklusive des WALs wurde im Festplattenabbild auf dem Desktop abgelegt und kann somit forensisch analysiert werden.

Der Ablageort der Datenbank im Festplattenabbild ist wie folgt:

```
C:\Users\%Benutzer%\Desktop\panama.android.notes\databases
```

### 6.1 Belkasoft (Szenario Memorix)

Da es sich bei der Android-App um keine Windows-spezifische Applikation handelt und die Applikation von Belkasoft zum Finden von Artefakten nicht unterstützt wird, erscheinen Inhalte nicht in der Übersicht.

Um die Datenbank zu analysieren, navigieren wir mittels „File System“-Ansicht zum oben genannten Pfad und öffnen die Datei notes\_db mit Hilfe des SQLite Viewers.

Auch hier erhalten wir über die Properties erste forensisch relevante Informationen:

Properties	
Page size:	4096
Number of pages:	73
Number of freelist pages:	0
Journal type:	Wal
Encoding:	UTF-8
MD5:	9E7419C0F8978AF8BE1E69741D3D994D
SHA1:	8BE513E0F09F2A514897F73A12EF5A7BA130268E

**Abbildung 6-1 Datenbankeigenschaften von notes\_db in Belkasoft**

Die Ansicht „Unallocated space“ ist in diesem Fall leer.

In der Tabelle „sequence“ gibt es den Schlüssel „entry\_order“ mit dem Wert 134. Es ist davon auszugehen, dass dieser Wert hochzählt, sofern ein neuer Eintrag in der Tabelle „entry“ angelegt wird. Somit wären fehlende Zahlenbereiche ein Indikator für gelöschte Notizen.

Dies kann über die Tabelle „entry“ mittels Spalte „\_order“ verifiziert werden:

Items: 350 Number of journal records: 222				
Data		Structure		
<input type="checkbox"/>	Record type	id [Primary key]	<input type="checkbox"/> _order	created_millis
<input type="checkbox"/>		85804bc3-4e41-416d-bc2f-4641bfa7aa29	1	9/17/2018 7:40:55 AM
<input type="checkbox"/>		5a1f1ac3-0213-46ee-873e-09c8db7ba0e5	2	9/17/2018 7:43:11 AM
<input type="checkbox"/>		77549a58-0023-4a09-940e-62e28ab488f6	3	9/21/2018 7:11:48 AM
<input type="checkbox"/>		27662e81-4a45-446c-9ed2-3dad11c26e08	4	9/21/2018 12:14:31 PM
<input type="checkbox"/>		9e0d5f9b-ff05-4b09-b324-c1791aa4c6d9	5	9/24/2018 7:22:50 AM
<input type="checkbox"/>		4955d901-6f4c-47f0-8a52-437be3882ea6	6	9/27/2018 7:36:43 AM
<input type="checkbox"/>		4161e032-7cb2-4ce0-b926-3acbbbed3258b	7	9/28/2018 3:30:27 PM

**Abbildung 6-2 Anzeige der ersten Zeilen der Tabelle entry**

<input type="checkbox"/>		ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM
--------------------------	--	--------------------------------------	-----	----------------------

**Abbildung 6-3 Letzter Eintrag in der Tabelle entry**

Da der letzte Eintrag der Tabelle die Annahme bestätigt kann über das Fehlen von Nummern in der Spalte „\_order“ von gelöschten Einträgen ausgegangen werden. Zudem kann ebenso der Zeitraum der Erstellung beziehungsweise des gelöschten Eintrags vermutet werden:

ad1f3c3c-8524-48af-a2ff-d4a807483205	106	8/29/2022 9:35:26 PM
e4051b71-57e0-4aff-93f2-03943b4b94bc	108	9/2/2022 11:27:10 AM

**Abbildung 6-4 Fehlender Eintrag in der Tabelle entry**

In diesem Beispiel fehlt die Nummer 107. Aufgrund der angrenzenden Sequenznummern 106 und 108 ist daraus zu schließen, dass der Eintrag 107 nach dem 29.08.22 09:35 und vor dem 02.09.22 11:27 angelegt worden ist. Es ist dazu kein Eintrag im WAL vorhanden, da dieser sonst angezeigt werden würde.

Eine weitere Möglichkeit ist das Einbeziehen der „Row id“-Spalte. Hier sind Lücken in der Nummerierung feststellbar. Allerdings scheint es, dass die Applikation die Spalte manipuliert. Daher ist diese Information, ohne den Programmcode zu kennen, unter dem Gesichtspunkt der IT-Forensik mit Vorsicht zu betrachten.

Als Beispiel steht dafür der folgende Eintrag:

<input type="checkbox"/>	Row id	Record type	id [Primary ke	_order	created_millis	last_modified_millis
<input type="checkbox"/>	229		d5059a9a-46	110	10/11/2021 11:58:51 AM	2/3/2024 1:32:35 PM
<input type="checkbox"/>	221	Wal	d5059a9a-46	110	10/11/2021 11:58:51 AM	2/3/2024 1:02:36 PM

**Abbildung 6-5 Unterschiedliche ROWIDs für den gleichen Eintrag**

Es handelt sich um den gleichen Eintrag mit gleichem Erstellungs-, aber unterschiedlichem Änderungsdatum. In der Datenbank hält der Eintrag die „Row id“ 229 und im WAL wird er mit 221 geführt. Das würde auf eine Erhöhung der ROWID schließen lassen, zieht sich aber nicht schlüssig durch die restliche Datenbank. Es gibt andere Einträge, die in Datenbank und WAL identisch sind (bezogen auf Inhalt, \_oder, Erstellungs- und Änderungsdatum), aber unterschiedliche ROWIDs führen.

Mit Hilfe des WALs können wir nun von Notiz-Einträgen eine Änderungshistorie herstellen. Diese reichen SQLite-bedingt (programmierbedingter Checkpoint) beziehungsweise Anwender-bedingt (z.B. Smartphone-Neustart) nur eine gewisse Zeit zurück. Als Beispiel eignet sich der Eintrag mit dem „title“ „Smalltalk“:

cord type	id [Primary key]	_order	created_millis	last_modified_millis
	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	2/6/2024 8:24:59 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	2/6/2024 8:24:59 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	2/6/2024 8:24:59 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	2/6/2024 8:24:59 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/5/2024 10:16:15 AM
Wal	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	134	9/27/2022 7:47:36 PM	3/6/2024 3:02:02 PM

Abbildung 6-6 Übersicht der verfügbaren Versionen des Eintrags Smalltalk

Anhand der Spalte last\_modified\_millis ist zu entnehmen, dass wir den Inhalt der Notiz „Smalltalk“ zu vier verschiedenen Zeitpunkten auswerten können.

Tabelle 6-1 Wiederhergestellte Versionen des Eintrags Smalltalk

Datum	Inhalt
06.02.2024 08:24:17	[{"id":"5b8f9bc6-26d0-4060-b609-8356df17f773","text":"Steuer (Rückzahl.)","checkable":true,"checked":false}, {"id":"dcee5c74-7803-49d9-a24f-0a2c80edbab5","text":"xxxxxxx","checkable":true,"checked":false}, {"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"xxxxxxxxxxxxx","checkable":true,"checked":false}]
06.02.2024 08:24:59	[{"id":"810f509f-02d6-4185-89a0-57fc7c2ea6e9","text":"xxxxxxxxxxxxx","checkable":true,"checked":false}, {"id":"525e1d0c-5bb4-4858-8b2a-68a6f2308a87","text":"xxxxxxx","checkable":true,"checked":false}, {"id":"bfe675eb-54c1-4d3b-b3ef-8217693978a7","text":"xxxxxxxxxxxxx","checkable":true,"checked":false}, {"id":"5b8f9bc6-26d0-4060-b609-

	<code>8356df17f773","text":"","checkable":true,"checked":false}]</code>
05.03.2024 10:16	<code>[{"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"","checkable":true,"checked":false}]</code>
06.03.2024 15:02	<code>[{"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"xxxxxxxxxxx","checkable":true,"checked":false},  {"id":"0b839bde-cb4a-465f-8cc0-19ac8cd2ee60","text":"xxxxxxxxxxx","checkable":true,"checked":false},  {"id":"63ec3d72-dcd1-4cbd-a5b2-d2060aef6a60","text":"xxxxxxxxxxx","checkable":true,"checked":false},  {"id":"b6f42cc0-d168-4ef3-ba6c-03266c36e9c0","text":"","checkable":true,"checked":false}]</code>

Wir können die Notiz nicht im Urzustand (Erstellungsdatum 27.09.2022, 19:47 Uhr) betrachten, aber dennoch gelöschte Inhalte wiederherstellen beziehungsweise eine Änderungshistorie abbilden.

## 6.2 Axiom (Szenario Memorix)

Um die Datenbank in Axiom analysieren zu können, müssen wir, wie auch bei Belkasoft, über die „File system“-Ansicht zum Ablage-Pfad navigieren. Nach einer Prüfung steht für die Datei „notes\_db“ der „SQLite Viewer“ zur Verfügung. Sofern wir den WAL der Datei, notes\_db-wal, markieren wird der Inhalt im „Preview“-Fenster als durchsuchbarer Text angezeigt. Warum eine Durchsuchung über das Previewfenster in diesem Fall möglich ist, ist nicht nachvollziehbar.

Was bei einer ersten Prüfung der Datenbank über den „SQLite Viewer“ ins Auge fällt ist die Diskrepanz zwischen der Sequenznummer (134) und der Zeilenanzahl der Tabelle „entry“ (128). Wie auch bereits bei der Analyse mittels Belkasoft festgestellt, fehlen hier Einträge. Mit Hilfe der im Anhang beschriebenen Abfrage können wir uns die fehlenden Nummern anzeigen lassen:

FIND			BUILD QUERY			EXPORT		
#	x	▼						
1	20							
2	71							
3	72							
4	73							
5	74							
6	107							

Abbildung 6-7 Ergebnis der SQL-Abfrage

Danach ist es möglich mit einer weiteren Abfrage (siehe 9.2) das Datum der Erstellung der Einträge einzugrenzen. Als Beispiel fragen wir die Erstellungszeit für die Einträge 106 und 108 ab:

FIND			BUILD QUERY			EXPORT			CLEAR FILTERS		
#	created	▼	_order	▲	▼						
1	2022-08-29 19:35:25		106								
2	2022-09-02 09:27:10		108								

Abbildung 6-8 Ergebnis der SQL-Abfrage um Erstellungszeitraum einzugrenzen

Somit ist davon auszugehen, dass der Eintrag 107 zwischen dem 29.08.2022 19:35 Uhr und dem 02.09.2022 9:27 Uhr erstellt wurde.

Da der WAL von Axiom in diesem Szenario per „Preview“-Ansicht durchsuchbar ist, ist es mit geringem manuellem Aufwand möglich die verschiedenen Versionen eines Eintrags zu extrahieren.

Über eine Abfrage der „sqlite\_master“-Tabelle stellen wir fest, dass die Spalte „id“ der Tabelle „entry“ als Primärschlüssel fungiert. Wir können auch über einen Wert der Spalte „\_order“ suchen, allerdings erachten wir den längeren und durch eine Buchstaben-Zahlenkombination einzigartigen Primärschlüssel als die bessere Alternative für eine Stringsuche.

Wir extrahieren die ID zum Eintrag „Smalltalk“, suchen nach dieser und finden 39 Einträge in der WAL-Datei. Sofern wir die Suchergebnisse durchgehen, filtern wir die doppelten Einträge manuell und erhalten die folgenden einzigartigen Einträge:

Tabelle 6-2 Wiederhergestellter Inhalt der Notiz Smalltalk mittels Suche

Eintrag	Inhalt
1	<pre>ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8Smalltalk[{"id":"5b8f9bc6-26d0-4060-b609-8356df17f773","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"dcee5c74-7803-49d9-a24f-0a2c80edbab5","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"xxxxxxxxxxx","checkable":true,"checked":false}]</pre>
2	<pre>ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8Smalltalk[{"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"","checkable":true,"checked":false}]</pre>
3	<pre>ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8Smalltalk[{"id":"3014e18f-48ae-4799-8385-9b78376761d5","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"0b839bde-cb4a-465f-8cc0-19ac8cd2ee60","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"63ec3d72-dcd1-4cbd-a5b2-d2060aef6a60","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"b6f42cc0-d168-4ef3-ba6c-03266c36e9c0","text":"","checkable":true,"checked":false}]</pre>
4	<pre>[{"id":"810f509f-02d6-4185-89a0-57fc7c2ea6e9","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"525e1d0c-5bb4-4858-8b2a-68a6f2308a87","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"bfe675eb-54c1-4d3b-b3ef-8217693978a7","text":"xxxxxxxxxxx","checkable":true,"checked":false},{"id":"5b8f9bc6-26d0-4060-b609-8356df17f773","text":"","checkable":true,"checked":false}]</pre>

Das sind exakt die Einträge, welche auch mittels Belkasoft gefunden wurden. Allerdings fehlt die Information der Bearbeitungszeit („last\_modified\_millis“). Eine Extraktion direkt aus der Vorschau ist nicht möglich.

Es gibt jedoch eine manuelle aufwändige Möglichkeit, um die Bearbeitungszeiten zu extrahieren. Dafür starten wir eine manuelle Suche nach der ID mittels „Text and Hex“-Funktion. Hier nutzen wir die Möglichkeit der Stringsuche und werden auch fündig. Allerdings ist nicht einfach erschließbar in welcher Reihenfolge einzelne Zellen im WAL abgelegt werden. Um ein aufwändiges Überprüfen der Reihenfolge von Spalten zu umgehen, arbeiten wir mit bereits vorhandenen Informationen. Alle Einträge in der Datenbank haben eine Erstellungszeit. Die Information über die Erstellungszeit liegt in der Datenbank für jeden entsprechenden Eintrag ab. Rechnen wir den Wert in hexadezimal um, erhalten wir einen 6 Bytes großen Suchwert. Dieser Wert ist innerhalb des Eintrags zu suchen und wird auffindig gemacht. Da die Spalten „Erstellt“ „zuletzt geändert“ in der Ansicht aufeinander folgen extrahieren wir die folgenden 6 Bytes und überführen diese in einen Dezimalwert. Der Dezimalwert ist wiederum in Datum und Uhrzeit umzurechnen:

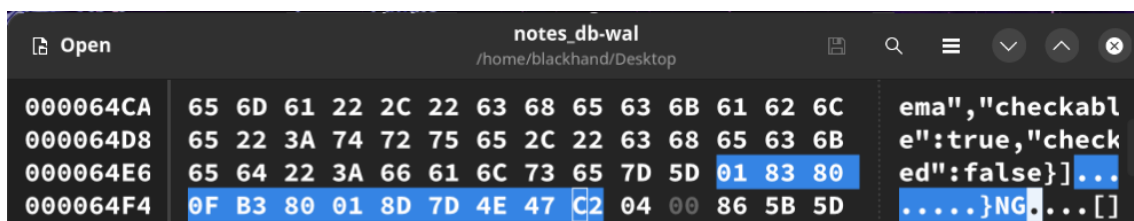


Abbildung 6-9 Ausschnitt der gefundenen Werte in einem Hexeditor

	Linux Epoch	Hexwert	Datum / Uhrzeit
erstellt	1664300856192	0x01 83 80 0F B3 80	27.09.22 17:47 Uhr
bearbeitet	1707204298690	0x01 8D 7D 4E 47 C2	06.02.24 07:24 Uhr

Tabelle 6-3 Zusammenfassung der gefundenen Werte

### 6.3 Frei verfügbare Software (Szenario Memorix)

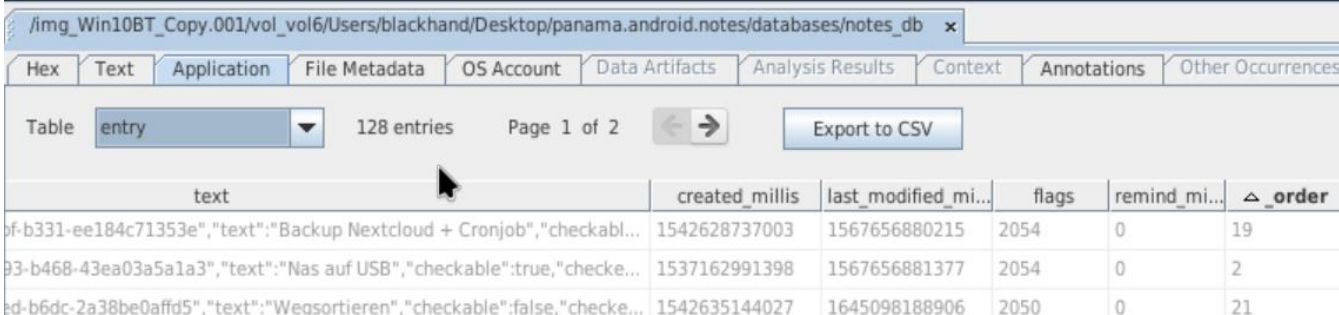
Auch in diesem Szenario analysieren wir mittels frei verfügbarer Software die auf



dem Desktop abgelegte Datenbank der Notiz-App.

### 6.3.1 Autopsy / Sleuthkit

Für die Datenbank der Notiz-App kann der in Autopsy eingebaute Viewer genutzt werden. Wie bereits unter Axiom festgestellt, können ohne WAL keine weiteren Artefakte wie gelöschte oder geänderte Einträge extrahiert werden. Das Einzige, was manuell möglich ist, ist das Finden von nicht mehr vorhandenen „\_order“ Nummern in der Tabelle „entry“:

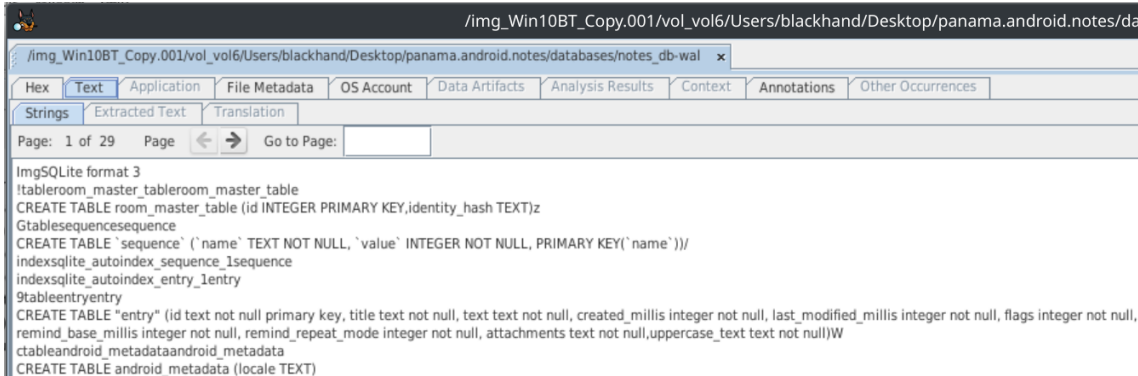


text	created_millis	last_modified_millis	flags	remind_millis	_order
bf-b331-ee184c71353e", "text": "Backup Nextcloud + Cronjob", "checkabl...	1542628737003	1567656880215	2054	0	19
93-b468-43ea03a5a1a3", "text": "Nas auf USB", "checkable": true, "checke...	1537162991398	1567656881377	2054	0	2
ed-b6dc-2a38be0affd5", "text": "Wegsortieren", "checkable": false, "checke...	1542635144027	1645098188906	2050	0	21

Abbildung 6-10 Anzeige einer Lücke in der Spalte \_order

Eine Automatik in Autopsy entfällt aus Mangel der Möglichkeit von SQL-Abfragen in der Datenbank. Allerdings ist es, wie vorangegangen erwähnt, möglich manuell die wahrscheinlichen Erstellungszeiten für eine Notiz zu erheben, indem der zeitliche Zwischenraum der benachbarten Notizen beachtet wird.

Da auch WALs nicht von Autopsy verarbeitet werden bleibt hier das manuelle Durchsuchen von extrahiertem Text, der sich in diesem Fall über 29 Seiten erstreckt. Eine Suchfunktion ist auch nicht gegeben:



```

SQLite format 3
itableroom_master_tableroom_master_table
CREATE TABLE room_master_table (id INTEGER PRIMARY KEY,identity_hash TEXT)z
Gtablesequencesequence
CREATE TABLE `sequence` (`name` TEXT NOT NULL, `value` INTEGER NOT NULL, PRIMARY KEY(`name`))
indexsqlite_autoindex_sequence_1sequence
indexsqlite_autoindex_entry_1entry
9tableentryentry
CREATE TABLE "entry" (id text not null primary key, title text not null, text text not null, created_millis integer not null, last_modified_millis integer not null, flags integer not null, remind_base_millis integer not null, remind_repeat_mode integer not null, attachments text not null,uppercase_text text not null)W
ctableandroid_metadataandroid_metadata
CREATE TABLE android_metadata (locale TEXT)

```

Abbildung 6-11 Auszug aus der Textansicht des WALs

Die einzige Möglichkeit hier produktiv Informationen zu finden, ist alle 29 Seiten

manuell in ein neues Dokument zu überführen und dort nach Artefakten mittels der „id“ aus der Datenbank „entry“ zu suchen. Diesen Weg halten wir allerdings für nicht praktikabel.

### 6.3.2 FQLite

Vor der Analyse der Datenbank der Notiz-App mit FQLite, extrahieren wir beide Dateien mit Autopsy aus dem Abbild.

Was beim Einlesen der Datenbank auffällt, ist dass im WAL der Ordner „\_\_Freelist“ nicht leer ist. Daher ist bei der Datenbank davon auszugehen, dass „auto\_vacuum“ nicht aktiv ist. Hier sehen wir einen guten Ansatz nach den zuvor vermissten Einträgen (20, 71, 72, ...) zu suchen, bleiben aber erfolglos.

In der Tabelle „entry“ des WALs können wir gezielt nach veränderten Datensätzen suchen. FQLite listet alle Datensätze inklusive der Versionsnummer auf. Sofern wir über die Spalte „Status“ nach „update“ suchen, finden wir die alle Einträge die in diesem WAL aktualisiert worden sind:

No.		Offset	PLL	HL	commit	dbpage	w...	salt1	salt2	id	title
80	004. version update	264347	519	14	true	72	64	582029881	497273176	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	Smalltalk [
83	005. version update	272670	436	14	true	72	66	582029881	497273176	ad17ed73-6dfb-44b0-8013-9eb9a1cecdf8	Smalltalk [
92	005. version update	284683	343	15	true	72	69	582029881	497273176	7bcd7c82-a5b1-42bd-98c9-b21eccbe7808	Klamottengrößen
96	006. version update	292917	349	15	true	72	71	582029881	497273176	7bcd7c82-a5b1-42bd-98c9-b21eccbe7808	Klamottengrößen

Abbildung 6-12 gefilterte Einträge im WAL durch FQLite

Auch hier finden wir den Eintrag „Smalltalk“ mit zwei Aktualisierungen. Ein weiterer Hinweis ist die Nummerierung der „Status“-Spalte. Die beiden Einträge für Smalltalk wurden zum vierten beziehungsweise fünften Mal editiert. So können häufig frequentierte Einträge identifiziert werden.

Mit einem gewissen Aufwand ist es auch mit FQLite möglich eine Änderungshistorie eines Eintrags innerhalb der Datenbank nachzuvollziehen. Dafür muss allerdings beachtet werden, dass alle Tabellen miteinbezogen werden. Am Beispiel des Smalltalk-Eintrages sind Einträge aus der eigentlichen Datenbank zu berücksichtigen. Danach müssen die Einträge aus dem WAL der Tabelle entry extrahiert und anschließend der Freelist-Container auf weitere Einträge geprüft werden. Nach einer Prüfung finden wir auch mit FQLite vier

Versionen der Notiz Smalltalk.

Des Weiteren liest FQLite den Header des WALs aus. Dort sind sechs Checkpoints vermerkt. Sofern wir uns die Übersicht der Checkpoints ansehen, können wir feststellen, dass zwei Checkpoints nicht mehr aufgeführt sind. Somit sind aus diesen Checkpoints keine Artefakte im WAL zu finden.

Mittels FQLite und manueller Nacharbeit ist es über die Saltwerte, die den einzelnen Checkpoints zugeordnet sind, möglich einzelne Datensätze in bestimmte Zeiträume einzuordnen.

Es werden alle Zeilen der Tabelle „entry“ manuell in eine Textdatei exportiert. Ein Versuch über die in FQLite angebotene Option „Export Node“ schlug fehl. Die Daten werden kommasepariert abgelegt und müssen für den Import in Excel bereinigt werden.

Ist dies passiert können die Zeitangaben umgewandelt und die Liste auf einzelne Salt1-Werte gefiltert werden. Nun ist es möglich die zuletzt geänderten Einträge der einzelnen Checkpoints zu identifizieren und Zeiträume festzulegen:

**Tabelle 6-4 Zeiträume der einzelnen Checkpoints**

Saltwert	Zeitraum
582029879	Alles vor dem 21.01.24
582029880	Zwischen dem 21.01.24 und 03.02.24
582029881	Zwischen dem 03.02.24 und 10.02.24
582029882	Zwischen dem 10.02.24 und 18.02.24

### 6.3.3 Pythonskripte

Auch hier haben wir versucht in Verbindung mit bring2lite, wal-crawler und unserem selbsterstellten Skript Informationen aus dem WAL zu extrahieren und über ein Einfügen in die Datenbank besser durchsuchbar zu machen.

Allerdings scheitern beide Skripte am WAL der Datenbank. Hier treten offenbar



## 7 Vergleich der eingesetzten Software

Nachdem wir die Softwares vorgestellt und eingesetzt haben und Daten extrahieren konnten, vergleichen wir die von uns eingesetzten Softwares. Wir beschränken uns auf die verwendeten forensischen Softwarepakete und lassen die Pythonskripte außen vor, da diese nur für einen speziellen Teilaspekt erstellt und eingesetzt worden sind.

### 7.1 Objektive Kriterien

Wir haben uns für die folgenden Kriterien zur objektiven Bewertung entschieden: Unterstützte Betriebssysteme, kostenpflichtig oder kostenlos, Art bzw. Umfang der Dokumentation, Möglichkeit der SQL-Abfrage, Unterstützung für temporärer Dateien, das Aufführen von Freeblocks und -lists und abschließend das Auslesen, die Anzeige von forensisch relevanten Pragma-Optionen.

Somit prüfen wir zuerst auf welchem Betriebssystem die Software lauffähig ist und stellen dann fest, ob sie kostenpflichtig oder kostenlos ist.

Danach stellen wir die Art und den Umfang der Dokumentation vor. Sind alle Funktionen nachvollziehbar erklärt?

Als nächstes kommen wir zu den beiden wichtigeren Punkten:

Zum einen die Implementierung der SQL-Abfrage an die Datenbank. Dies bietet dem Forensiker eine Möglichkeit Zusammenhang von Daten festzustellen oder Lücken in größeren Datensätzen zu finden.

Zum anderen und der wichtigste Punkt, welcher in dieser Thesis herausgearbeitet worden ist, die Unterstützung für Rollback-Journals und Write-Ahead-Logs der Datenbank. Sofern diese erhalten sind, ist die Chance auf das Finden und Wiederherstellen von gelöschten oder geänderten Datensätzen weitaus größer. Daher sollten diese komfortabel analysierbar sein.

Anschließend betrachten wir die Anzeige des Inhalts von Freelists und Freeblocks innerhalb der Dateien, welche im Allgemeinen nicht mehr verwendete Seiten beinhalten oder kurze freie Blöcke zwischen einzelnen Zeilen innerhalb

einer Seite / Tabelle abbilden.

Zuletzt betrachten wir die Anzeige oder die Abfrage von forensisch relevanten Pragma-Optionen wie „secure\_delete“ und „auto-vacuum“.

Um die Bewertung in Zahlen auszudrücken, werden wir die einzelnen Kriterien nach Wichtigkeit gewichten und aufsummieren.

Unterstützt die Software einen der oben genannten Punkte erhält sie dafür einen Wert zwischen 1 und 0. Dieser Wert wird mit dem Faktor der Gewichtung multipliziert. Alle Werte ergeben aufsummiert schlussendlich eine Punktzahl. Je höher der Wert, desto besser hat die Software abgeschnitten.

Die vollständige Tabelle mit den Gewichtungen für die Kriterien und die Bewertungen inklusive des Ergebnisses ist im Anhang 9.5 zu finden.

## 7.2 Objektive Kriterien (Tabelle)

Tabelle 7-1 Bewertung der eingesetzten Software nach Bewertungskriterien

Software	Betriebssystem	Kostenpflichtig	Kostenlos	Dokumentation	SQL-Abfragen	WAL / Journal-Unterstützung	Freeblocks	Freelists	Pragma-Abfrage / -anzeige
Belkasoft	Windows	X		Gut	Nein	Ja	Ja	Ja	Nein
Axiom	Windows	X		Gut	Ja	Teilweise	Nein	Nein	Nein
Autopsy	Windows, Linux, MacOS		X	ausreichend	Nein	teilweise	Nein	Nein	Nein

FQLite	Windows, Linux, MacOS		X	man gelh aft	Nein	Ja	Nein	Ja	Nein
--------	-----------------------------	--	---	--------------------	------	----	------	----	------

### 7.3 Bewertung der eingesetzten Software

#### 7.3.1 Belkasoft

Belkasoft zeichnet sich durch eine gute Dokumentation in Form von PDFs und Videos des Herstellers aus.

Zum eigentlichen Thema Forensik in SQLite-Datenbanken gibt es einen recht umfangreichen Eintrag [14]. Dort wird angegeben, dass die Eigenschaften Freeblocks und Freelists unter dem Abschnitt „unallocated Space“ summiert sind. In unserem Experiment konnte das nicht eindeutig nachvollzogen werden.

Abfragen über SQL an die Datenbank sind nicht möglich.

Dafür unterstützt Belkasoft WALs und Rollback-Journals und bereitet diese in der grafischen Oberfläche sichtbar auf. Eine Anzeige für weitere forensisch relevante Pragma-Optionen, außer dem Journaltyp, wäre wünschenswert.

#### 7.3.2 Axiom

Auch hier steht eine reichhaltige Dokumentation des Herstellers zur Verfügung. Wie auch bei Belkasoft sind es PDFs und Videos. Der Eintrag zu SQLite ist allerdings weniger umfangreich. Dafür wurde in der neuesten Version des eingesetzten Analyseteils die Möglichkeit für SQL-Abfragen geschaffen, die allerdings eine Abfrage von Pragma-Optionen nicht ermöglicht.

Eine Anzeige, welche Freelists oder Freeblocks innerhalb der SQLite-Datenbank aufführt, ist nicht implementiert.

Die temporären Dateien werden nicht nativ unterstützt, sind aber immerhin mit einem Text-Viewer manuell und überschaubarem Aufwand durchsuchbar.

#### 7.3.3 Autopsy

Ein Vorteil von Autopsy zu den bereits genannten Softwarepaketen ist die

Verfügbarkeit auf allen größeren Betriebssystemen. Die Dokumentation ist für eine OpenSource-Software reichhaltig, sofern man gewillt ist die manuelle Suche im Internet zu tätigen.

Ansonsten fehlt der Software in Bezug auf eine forensische Untersuchung von SQLite-Datenbanken vieles.

Wir haben keine Unterstützung von WALs oder Rollback-Journals. Das manuelle Durchsuchen der temporären Dateien gestaltet sich schwierig da der Inhalt nur seitenweise angezeigt wird und keine Suchfunktion implementiert ist.

Eine Abfrage über SQL ist nicht gegeben und der eingebaute SQLite-Viewer teilt die vorhandenen Tabelleneinträge auf mehrere Seiten auf, was eine zusammenhängende Suche erschwert. Zudem wurden nicht alle Einträge aus der Datenbank in die Übersicht der gefundenen Artefakte übertragen. Pragma-Optionen werden ebenfalls nicht dargestellt.

Die hinzugefügten Erweiterungen für das Wiederherstellen von gelöschten Einträgen innerhalb der Datenbank haben in beiden Experimenten nicht funktioniert bzw. deren Funktion war nicht nachvollziehbar.

#### **7.3.4 FQLite**

FQLite punktet wie Autopsy mit einer Verfügbarkeit auf allen größeren Betriebssystemen.

Die Dokumentation zur Software ist äußerst dürftig. In der bestehenden Anleitung werden nur die absoluten Grundfunktionen erklärt.

Dafür werden beide Arten der temporären Dateien (WAL und Rollback-Journal) von der Software verarbeitet. Die Software erlaubt das Sichten von Datensätzen in Datenbanken und temporären Dateien in einer excelartigen Ansicht. Mittels Filter über eine spezifische Spalte lassen sich Suchergebnisse eingrenzen.

Zudem werden die Freelists aufgeführt, aber keine Freeblocks, zumindest war dies nicht klar ersichtlich.

Des Weiteren gibt es viele Informationen wie eine Checkpoint-Liste und aufbereitete Header-Informationen. SQL-Abfragen und damit das Abfragen von gesetzten Pragma-Optionen sind nicht möglich.



## 7.4 Zusammenfassung und Bewertung

Die beste Bewertung erhält in unserer Thesis die Software von Belkasoft. Durch eine gute Dokumentation und der Unterstützung der temporären Dateien und Anzeige von nicht zugewiesenem Speicher (inklusive Freeblocks und Freelists) konnten wir damit die Datenbank genau analysieren. Auf der negativen Seite steht die fehlende Möglichkeit zur SQL-Abfrage. Wir gehen davon aus, dass dies eine Analyse von größeren Datenbanken erschwert.

Zudem ist Belkasoft nur unter Windows verfügbar. Zumindest wurde uns nur eine Windowsversion zur Verfügung gestellt und auf der Produktseite sind keine Hinweise auf andere Betriebssysteme zu finden.

FQLite folgt auf Belkasoft. Zwar ist die Dokumentation sehr dürftig, aber durch die Unterstützung und das Auslesen der temporären Dateien war eine gute Analyse der Datenbanken möglich. Auch ist das Aufführen der Freelists als Pluspunkt zu sehen. Des Weiteren ist ein positiver Punkt die breite Unterstützung für verschiedene Betriebssysteme. Negativ ist hier der fehlende Support für SQLite-Abfragen.

Danach sehen wir Axiom in der Bewertungsskala. Es fehlt an nativer Unterstützung für die WAL und Rollback-Journals aber immerhin können die Dateien mit gewissem manuellem Aufwand untersucht werden. Allerdings fehlt hier die Unterstützung beziehungsweise die Anzeige von Freeblocks und Freelists gänzlich. Weiterhin gibt es, wie bei Belkasoft, die Software nur für Windows. Einen weiteren positiven Punkt sehen wir in der Möglichkeit zur SQL-Abfrage und dies ist ein Alleinstellungsmerkmal.

An letzter Stelle steht Autopsy. Hier mangelt es bei der Unterstützung für SQLite-Datenbanken. Allein die einfache Auflistung von Inhalten in der Datenbank ist in Seiten unterteilt und so schwierig zu durchsuchen. Es gibt keine Suchfunktion für die manuelle Analyse von temporären Dateien. Diese sind wiederum auch in einzelne Seiten zerlegt. Die Erweiterungen haben nicht funktioniert und in der Übersicht fehlten Seiteneinträge, die in der Datenbank ersichtlich sind. Vorteilhaft ist der Opensource-Charakter der Software. Unterstützung kann über das Forum oder eine allgemeine Internetsuche erhalten werden. Auch ist Autopsy für alle größeren Betriebssysteme verfügbar.

## 8 Diskussion

In der vorliegenden Arbeit wurde mit Hilfe von kommerzieller und frei verfügbarer forensischer Software untersucht, ob und wann Datensätze oder Artefakte aus SQLite-Datenbanken wiederhergestellt werden können. Aufgrund verschiedener fehlender Optionen in der verwendeten forensischen Software konnten Qualitätsunterschiede im Umgang mit Abfrage, Ansicht und Wiederherstellung von Datensätzen festgestellt werden. Des Weiteren hat sich, gezeigt dass die Pragma-Optionen „auto-vacuum“ und „secure\_delete“ einen messbaren Einfluss auf die Wiederherstellung haben.

Trotzdem war es uns möglich einige forensisch relevante Ergebnisse und Daten zu finden und wiederherzustellen oder zumindest Folgerungen zuzulassen.

Im Folgenden werden wir die Ergebnisse näher einordnen.

### 8.1 Bewertung der Ergebnisse

#### 8.1.1 Firefox

Wie mit dem „DB Browser for SQLite“ festgestellt, ist bei Firefox das Pragma „secure\_delete“ für die Datenbank places.sqlite und favicons.sqlite aktiv. Im Normalfall kann dieses Pragma mit Hilfe des WALs umgangen werden. Wie aus unserer Analyse sichtbar wird, hilft zumindest bei dieser Datenbank, kein Write-Ahead-Log. Es scheint, dass Firefox alle Spuren einer Webseite aus dem Verlauf und somit aus der Datenbank löscht oder zumindest die bei einer Löschung entstehenden Lücken in der Datenbank durch SQLite sofort über den „vacuum“-Befehl auffüllen und damit verschwinden lässt.

In Belkasoft konnte die Datenbank und das zugehörige Write-Ahead-Log zwar ausgelesen, musste aber aufgrund fehlender Unterstützung von SQL-Abfragen auf manuellem Wege ausgewertet werden.

Weiterhin mussten Zeiten umständlich in Excel auf ein für Menschen lesbares Format umkonvertiert werden. Zumindest für die Hauptdatenbank des Verlaufs konnte nur aufgrund von fehlenden IDs auf ein nicht mehr vorhanden sein von

Datensätzen geschlossen werden. Dies ist bei kleinen Datenbanken schnell ersichtlich, bei größeren Datensätzen ist eine Abfrage über SQL wünschenswert.

Mittels Datenbank favicons.sqlite, die in unserem Experiment eine geringe Größe aufwies, konnte mit wenig Aufwand identifiziert werden welches Vorschaubild nicht mit den Webseiten in places.sqlite korreliert. Bei größeren Datenbanken könnte dies ohne eine SQL-Abfrage unübersichtlich werden. Als Workaround gibt es die Möglichkeit des Exports der Daten und einer anschließenden Filterung (z.B. mittels Microsoft Excel).

Sofern wir auf Axiom schwenken, sehen wir den Nachteil einer forensischen Software, welche keine WALs verarbeiten kann in Verbindung mit einer restriktiven Handhabung von Löschungen der Einträge in einer SQLite-Datenbank. Wir haben in Axiom zwar die Möglichkeit der Durchsuchung des WALs mittels „Text and Hex“-Fenster, aber die Suchfunktion ist begrenzt und erlaubt auch keine Regex-Ausdrücke. Mittels Regex wäre es zumindest möglich nach Zeichenfolgen im Webseitenformat zu suchen, um nicht vorhandene Seiten zu finden. Ein manueller Brute-Force-Ansatz besteht aus dem Suchausdruck „https://www“. Damit werden aber alle Verlinkungen gefunden, was selbst in kleineren Datenbanken als aufwändig zu beschreiben ist.

Das mit Belkasoft über den WAL gefundene Vorschaubild beziehungsweise die damit verbundene Web-Adresse aus der zweiten Datenbank kann nicht gefunden werden.

Immerhin haben wir mit der eingebauten Möglichkeit zur SQL-Abfrage bei größeren Datenbanken eine komfortable und übersichtliche Tabelle der fehlenden ROWIDs.

Auch ist es möglich uns die „Create Table“-Statements der „sqlite\_master“-Tabelle anzusehen, um damit auf die Struktur, Primär- und Fremdschlüssel der Datenbank zuzugreifen. Diese Information kann beim Erstellen von SQL-Abfragen behilflich sein. An diesem positiven Punkt ist aber Kritik anzubringen. Die Anzeige des Ergebnisses der Abfrage wird in der grafischen Oberfläche gekürzt dargestellt und muss erst in eine Datei exportiert werden, um den vollständigen Text lesen zu können.

Kommen wir zur frei verfügbaren Software und fangen mit dem Sleuthkit /

Autopsy:

Autopsy kann zwar per se mit SQLite-Datenbanken umgehen und stellt dafür einen eingebauten Viewer zur Verfügung, hat aber keine komfortable Ansicht in welcher WALs analysiert werden können. Hier fehlt wie bei Axiom der native Support. Es ist nur eine manuelle Durchsuchung per Text bzw. Strings ohne Suchfunktion möglich, welche auf einzelne Seiten in der Ansicht begrenzt ist.

Eine SQL-Abfrage an die Datenbank ist nicht möglich, somit ist eine manuelle Durchsuchung auf Unstimmigkeiten oder Artefakte nötig.

Die zuvor gefundenen Python-Erweiterungen für Autopsy wurden eingebunden zeigen aber in unserem Experiment keine Wirkung.

Ebenso ist zu erwähnen, dass Autopsy nicht alle Webseiten in seiner Übersicht aufgeführt hat, welche in der Datenbank vermerkt sind. Damit finden wir in Autopsy nur die manuell festgestellten fehlenden „ROWIDs“ der einzelnen Datenbanken.

FQLite hingegen stellt eine gute Analysemöglichkeit für beide Dateitypen dar. Das Tool hat aber keine Möglichkeit der SQL-Abfrage oder eine Anzeige der Datenbank zu einem bestimmten Zeitpunkt mittels Checkpoint-Auswahl. Auch die Exportfunktion für Inhalte ist nicht komfortabel gelöst.

Es kann aber über alle Spalten in allen Tabellen gefiltert werden und so konnte festgestellt werden, dass die meisten Inhalte des WALs bereits in der Datenbank abgebildet waren.

Für einen detaillierten Überblick inklusive des Findens des Artefakts in der favicons.sqlite und somit als tiefgehende Ergänzung zu Autopsy ist es nutzbar.

Im letzten Abschnitt besprechen wir die Pythonskripte. Mit Hilfe von bring2lite war es möglich die einzelnen Einträge aus dem WAL zu extrahieren und mit unserem Skript konnten wir erfolgreich die Daten zurück in die eigentliche Datenbank importieren. Somit ist über den „DB Browser for SQLite“ eine detaillierte Durchsuchung mittels SQL möglich. Über die gezeigte SQL-Abfrage konnten die Einträge, welche wir bereits mit Belkasoft identifiziert haben, gefunden werden.

Der Nachteil an der primären Verwendung von bring2lite in Verbindung mit WALs ist allerdings das Nichtvorhandensein von Salt-Werten und ROWIDs der

einzelnen Zeilen. Würde diese Information mit extrahiert werden, könnten wir von einem Plus an Informationsgewinnung ausgehen.

Generell sehen wir diese Variante als unterstützende Zusatzoption zu den IT-forensischen Softwarepaketen.

Weiterhin konnten wir erfolgreich die Freeblocks der einzelnen B-tree-Seiten extrahieren, auch wenn diese keine weiteren Informationen enthalten dank der aktivierten „secure\_delete“-Option.

Zusammenfassend können wir festhalten, dass im Fall von Firefox mit jeder Software ein Fehlen von Einträgen feststellbar war. Über die zweite Datenbank favicons.sqlite konnten einzelne Webseiten identifiziert werden, die wahrscheinlich aus einem vorherigen oder gelöschten Verlauf stammen. Mit dem Einbeziehen der Spalte „expire\_ms“ konnte auch der Zeitpunkt des wahrscheinlich letzten Aufrufs der Seite identifiziert werden.

Trotz Write-Ahead-Log waren kaum forensische Details aus der Verlaufsdatenbank wiederherstellbar.

Zum einen können wir secure\_delete als Grund für die geringe Ausbeute an Artefakten ansehen. Es sind Freeblocks in Datenbank und WAL vorhanden aber diese bestehen leeren Inhalten.

Zum anderen ist die Option „auto-vacuum“ für die Verlaufsdatenbank places.sqlite deaktiviert, aber es waren keine Freelists zu finden. Wir gehen davon aus, dass die Datenbank nicht ausreichend mit Inhalten befüllt und gelöscht wurde, um komplette Seiten als frei verfügbar zu markieren. Wir sehen auch die mit 32768 Bytes gewählte Seitengröße als weiteren Grund.

Eine Vermutung ist zudem die Programmierung von Firefox im Umgang mit dem Schreiben von Informationen in die Datenbank. Mit FQLite stellen wir fest, dass die meisten Einträge aus Datenbank und WAL übereinstimmen. Somit ist von einer häufigen Checkpoint-Tätigkeit auszugehen.

### **8.1.2 Memorix**

Mit Belkasoft und dem vorhandenen Write-Ahead-Log können in der vorliegenden Datenbank Einträge in verschiedenen Versionen wiederhergestellt

---

und daraufhin Änderungen im zeitlichen Verlauf nachvollzogen und analysiert werden.

Auch hier konnten ohne SQL-Abfrage Lücken in der Vergabe der „\_order“-ID festgestellt werden. Diese Lücken konnten über den Inhalt des WALs nicht aufgefüllt werden.

Die fehlenden „ids“ in der Spalte „\_order“ konnten nicht über den WAL aufgefüllt werden. Die Option „unallocated space“ brachte keine weiteren Erkenntnisse.

Bei Axiom konnte mittels SQL-Abfrage komfortabel die Lücken der Spalte „\_order“ ermittelt werden. Auch ist es in diesem Fall möglich mittels Textsuche und manuellem Aufwand im WAL die unterschiedlichen Versionen von Einträgen zu identifizieren. Wird weiterer Aufwand betrieben können ebenso Zeit und Datumsinformationen zu den einzelnen Einträgen ermittelt werden.

Schwenken wir zur Software um, welche frei verfügbar ist und starten erneut mit Sleuthkit / Autopsy.

In Autopsy können wir uns den Inhalt der Datenbank ansehen und auch manuell Lücken in der Spalte „\_order“ identifizieren. Mit sehr viel Aufwand wäre es möglich einzelne Versionen von Einträgen über die Textansicht der WALs zu sichten. Allerdings gibt es in der Textansicht keine Suchfunktion

Über FQLite haben wir wieder gute Möglichkeiten die Versionen einzelner Einträge über Filter zu sichten und extrahieren. Freelists der Datenbank sind dort einsehbar, bringen uns aber im vorliegenden Experiment nicht weiter. Somit ist zu vermuten, dass der letzte Checkpoint von der Applikation nicht durchgeführt worden ist, da sonst die Freelists durch das Setzen von „auto\_vacuum“ auf den Wert 1 abgeschnitten werden.

Mittels des beschriebenen Exports und der angezeigten SALT-Werte ist es möglich eine Timeline über die Änderung alle Einträge der Datenbank zu erstellen. Dies wäre in einem größeren forensisch relevanten Kontext unter Umständen von Nutzen. In unserem Versuch konnten wir so die einzelnen Änderungszeiträume bestimmen.

Die Skripte zum Auslesen des WALs halfen hier nicht weiter. Somit ist ein Zusammensetzen der vorliegenden Informationen aus beiden Dateien nicht

möglich. Das aktive „secure\_delete“ ermöglicht keine weiteren Informationen beim Auslesen der Freeblocks, allerdings lässt die Existenz von Freeblocks auf gelöschte Daten schließen.

Es ist festzuhalten, dass in dieser Datenbank die Werte der ROWID durch die Applikation manipuliert werden. Dies lässt SQLite unter bestimmten Umständen auch zu. Somit sollte bei einer forensischen Untersuchung ein solches Verhalten nicht ausgeschlossen werden und die ROWID nicht als verlässlicher Primärschlüssel angesehen werden.

Zusammenfassend können wir in diesem Szenario ermitteln, dass uns, im Gegensatz zu Firefox, der WAL eine größere Hilfe im Rekonstruieren von Einträgen der Datenbank ist. Mit beiden Softwareklassen (kostenpflichtig, OpenSource) konnten Veränderungen in Einträgen der Datenbank nachvollzogen werden. Je nach Softwareeinsatz musste manueller Aufwand bei der Rekonstruktion eingesetzt werden.

Weiterhin haben wir festgestellt, dass die Programmierung der Applikation im Hinblick auf die Verwendung der SQLite-Datenbank zu berücksichtigen ist. Eine Wiederherstellung von Daten aus den Freeblocks war trotz WAL nicht möglich.

## **8.2 Fazit**

Die forensische Untersuchung von SQLite-Datenbanken, das Finden und Wiederherstellen von geänderten oder gelöschten Datensätzen mittels kommerzieller und frei verfügbarer Software ist abhängig von einigen Faktoren.

Zuallererst ist das Programm zu nennen, das die Datenbank erzeugt und verwaltet. Je nachdem, welche Optionen (PRAGMA) gesetzt werden, sind die Möglichkeiten der Extraktion von Daten begünstigt oder eingeschränkt. Ein elementarer Bestandteil ist die Erhaltung des Rollback-Journals oder Write-Ahead-Logs. Damit besteht die Möglichkeit die „secure\_delete“-Funktion außer Kraft zu setzen. Dies wurde von Sanderson [22, p. 160] beschrieben und kann von uns für beide Szenarien nicht bestätigt werden.

Weiterhin sollte die genutzte forensische Software eine Unterstützung für das Einlesen von Journal-Dateien haben. Wie im oberen Punkt beschrieben, kann

dies bei einer forensischen Extraktion von großer Bedeutung sein. Die Unterstützung verliert dann an Gewicht, sobald in der Datenbank die Anti-Forensik-Optionen „secure\_delete“ und „auto\_vacuum“ deaktiviert sind. Dann ist immerhin eine Extraktion von gelöschten Zeilen oder nicht mehr benötigten Seiten über die Freelists möglich.

Ist keine Unterstützung der temporären Dateien implementiert so sollte doch eine Durchsuchung auf Textebene, wenn möglich mit der Unterstützung von Regex-Ausdrücken über eine Suchfunktion möglich sein.

Sofern eine Unterstützung der temporären Dateien vorhanden ist, ist es sinnvoll eine Auflistung der Checkpoints und Commits angezeigt zu bekommen. Hier ist es möglich eine Timeline für die Datenbank aufzubauen.

Wobei hier zu beachten ist, dass die Struktur des WALs und Rollback-Journal mindestens 10 Jahre nicht geändert worden ist. Es ist also kein ständiger Aktualisierungsaufwand für den Auslese-Algorithmus notwendig. Zudem hat es einen großen forensischen Mehrwert. Daher ist eine Unterstützung der temporären Dateien im Zuge des Aufwands eher gering. Auch sind bereits wissenschaftliche Arbeiten (z.B. Shu [9]) auf einen Algorithmus zum Auslesen von WAL eingegangen.

SQL-Abfragen innerhalb der Datenbank geben dem untersuchenden Forensiker ein weiteres Werkzeug zum Auffinden von gelöschten Datensätzen [22, pp. 267-269] an die Hand. Auch ist es möglich unterschiedliche Daten in Verbindung zu setzen. Als Beispiel führen wir den durchgeführten Reimport von Datensätzen aus einem Write-Ahead-Log in dieser Arbeit an (5.3.4).

Absolut wünschenswert ist eine Funktion, die automatisch die Daten der Datenbank und temporären Datei zu einem bestimmten Checkpoint-Zeitpunkt anzeigt. Dies ist im Forensic Toolkit for SQLite von Sanderson implementiert und bietet sicherlich einen Mehrwert. [22, p. 147]

Um ein Bild von der Möglichkeit der Wiederherstellung und des Findens von Datensätzen zu bekommen, sollte die Software zumindest forensisch relevante Pragma-Optionen auslesen und darstellen können. Für den eingesetzten Journal-Modus ist diese Information über den Datenbankheader möglich, für „secure\_delete“ und „auto\_vacuum“ nur per SQL-Abfrage.



Sofern es sich um eine Opensource-Software handelt, die ihre eigene Implementation der SQLite-Programmbibliothek mitbringt, kann dies auch über Prüfen des Sourcecodes geschehen. Dies wäre bei Firefox zutreffend.

Ein weiterer Punkt, welcher unbedingt Beachtung finden sollte, ist die Zugriffsmöglichkeit auf SQLite-Datenbanken. Wie im Abschnitt der Grundlagen erwähnt, liefert SQLite keine Benutzer- und Rechteverwaltung mit. Das heißt jeder Nutzer eines Geräts, der Zugriff auf das Dateisystem hat, kann die Daten manipulieren. Die Art und Reichweite des Zugriffs ist zwar je nach eingesetztem Betriebssystem erleichtert oder erschwert, sollte aber in das abschließende Untersuchungsergebnis mit einfließen.

Unsere Arbeit zeigt, dass die von uns eingesetzten Softwares die oben genannten Punkte nicht vollumfänglich erfüllen können. In der forensischen Analyse von SQLite-Datenbank besteht meist ein Teil der Arbeit aus manuellem Einsatz, zum Beispiel im Erstellen von eigenen Skripten, die eine tiefgehende Analyse und Extraktion von Daten ermöglichen. Des Weiteren konnten einige Techniken von Sanderson erfolgreich eingesetzt und bestätigt werden. Andere aufgestellte Thesen sind nachvollziehbar, konnten von uns aber nicht bestätigt werden.

### **8.3 Ausblick**

Laut der sehr gut dokumentierten SQLite-Webseite wird die Programmbibliothek bis heute weiterentwickelt. Allerdings handelt es sich hierbei überwiegend um Fehlerbehebungen und Erweiterungen von bestehenden Funktionen. Die letzten forensisch relevanten Änderungen waren das Hinzufügen des Wertes „fast“ für die Option „secure\_delete“ und liegt mit 2017 einige Jahre zurück.

Die Aktualisierungen der Unterstützung in Softwarepaketen wie Axiom für SQLite-Datenbanken (letzte Änderung ca. 2019 [15]) liegt bereits fünf Jahre zurück. Hier ist abzuwarten, ob in Zukunft eine Möglichkeit geboten wird eine tiefgehende Analyse durchzuführen.

Sofern man das Supportdokument von Belkasoft zu SQLite [14] analysiert, stellt man fest, dass die eingebetteten Screenshots von Version 1.11.

beziehungsweise 1.12. stammen. Durch die Versionshistorie der Softwarepakete und dem darin aufgeführten unterstützten Softwarestand für iOS lässt sich die letzte Aktualisierung auf vor ca. zwei bis drei Jahre datieren.

Vergleicht man den Funktionsumfang mit der Software von Sanderson, die im Buch [22] beschrieben wird, ist dieser sehr gering oder auch ausbaufähig, vor allem in Bezug auf Autopsy.

Bei dem aktuellen Stillstand in den beiden verwendeten Paketen bleibt es spannend, ob die gebotenen Implementierungen ausgebaut oder die Funktion über spezialisierte IT-forensische Applikationen abgedeckt werden müssen.

Weiterhin hält auch künstliche Intelligenz Einzug in die von uns verwendete Software. Sie wird zur Einordnung von Bildern in verschiedene Kategorien genutzt (Nacktheit, Gewalt, etc.). Ein Einbinden in die Analyse von SQLite-Datenbanken konnten wir nicht feststellen.

Auch bleibt abzuwarten, wie die Entwicklung von OpenSource auf dem Gebiet von SQLite weitergeht. Bring2Lite wird aktuell, ähnlich wie Undark, nicht mehr weiterentwickelt. Auch das von uns als Vorlage genutzte Pythonskript von DeGrazia [31] wurde seit Jahren nicht mehr weiterentwickelt.

Es scheint, als gäbe es auch hier auf dem Feld OpenSource einen Stillstand der vergleichbar mit der Entwicklung von SQLite und kommerzieller Software ist. Neue forensische Funktionen sind nicht zu finden.

Im Umkehrschluss heißt der Stillstand in der Entwicklung der Programmbibliothek SQLite sowie in den IT-forensischen Werkzeugen das getroffene Aussagen, Erkenntnisse und Workarounds, auch aus früheren wissenschaftlichen Arbeiten, weiterhin Bestand haben.

## 9 Anhang

### 9.1 Abfrage 1

```
with recursive cte(x) as (Select (Select Min(entry._order) as ColAlias0 from entry) AS ColAlias0
union all
select cte.x + 1 as colAlias1
from cte
where cte.x < (Select Max(entry._order) as ColAlias0 from entry))
select *
from cte
where cte.x not in (Select entry._order from entry)
```

### 9.2 Abfrage 2

```
select datetime(entry.created_millis/1000, 'unixepoch') AS created, _order
from entry
where _order = 106 or _order=108
```

### 9.3 Abfrage 3

```
SELECT page_url, page_url_hash
FROM moz_pages_w_icons_copy
WHERE page_url_hash
NOT IN (SELECT page_url_hash FROM moz_pages_w_icons)
```

### 9.4 Anhang A

Firefox wurde gestartet:

spiegel.de wurde aufgerufen

reddit.com wurde aufgerufen

In Firefox wurde nach dem Browser-Addon SQLITE gesucht

Youtube.com wurde aufgerufen

Die Maschine wurde pausiert, die aktuelle VDI gesichert, und am nächsten Tag wieder gestartet

In Firefox wurde der Browserverlauf des vorherigen Tages gelöscht

Hardwareluxx.de wurde aufgerufen und auf eine News mit dem Inhalt „Geekcom Mini Air12“ aufgerufen

Computerbase.de wurde aufgerufen und auf eine News mit dem Inhalt „Automobili Lamborghini“ aufgerufen

Ebay.de wurde aufgerufen und eine Suche mit dem Begriff „8tb western digital“ gestartet

## 9.5 Anhang Bewertungen

Software	Betriebssystem	Kommerziell	Opensource	Dokumentation	SQL-Abfrage	WAL / Rollback-Journal	Freeblocks	Freelists	Pragmatische Anzeige	Ergebnis
Gewichtung	0,3	0,3	0,6	0,5	0,8	1	0,5	0,5	0,5	
Belka	Windows 0,8	X 1	N/A 0	gut 0,8	Nein 0	Ja 1	Ja, aber nicht eindeutig abgegrenzt 0,8	Ja, aber nicht eindeutig abgegrenzt 0,8	Nein 0	2,74
Axiom	Windows 0,8	X 1	N/A 0	gut 0,8	Ja 1	teilweise, mit manuellem Aufwand verbunden 0,5	Nein 0	Nein 0	Nein 0	2,24
Autopsy	Windows, Linux, MacOS 1	N/A 0	X 1	ausreichend 0,4	Nein 0	möglich aber mit sehr hohem Aufwand verbunden 0,2	Nein 0	Nein 0	Nein 0	1,3
FQlite	Windows, Linux, MacOS 1	N/A 0	X 1	mangelhaft 0,2	Nein 0	Ja 1	Nein 0	Ja 1	Nein 0	2,5

## 9.6 Freeblock\_seeking.py

```
import os,struct,sys

def wal(f, output, filesize):
    f.seek(8)
    pagesize = struct.unpack('>L', f.read(4))[0]
    f.seek(16)
    salt1 = struct.unpack('>L', f.read(4))[0]
    salt2 = struct.unpack('>L', f.read(4))[0]
    firstline = str("Big Endian Wal-File, Salt1: {}, Salt2:
    {}\n".format(int(salt1),int(salt2)))
    output.write(firstline)
    offset = 32
    walframe = 1
    while offset < filesize:
        f.seek(offset)
        page_num = struct.unpack('>L', f.read(4))[0]
```

```

f.seek(offset + 8)
page_salt1 = struct.unpack('>L', f.read(4))[0]
f.seek(offset + 24)
ident_page = struct.unpack('>B', f.read(1))[0]
if ident_page == 13:
    first_free_block = struct.unpack('>H', f.read(2))[0]
    lineheader = ("Page: {},Walframe:
{}\n".format((int(page_num)),walframe))
    output.write(str(lineheader))
    while first_free_block != 0x0:
        f.seek(offset + 24 + first_free_block)
        freeblock = struct.unpack('>H', f.read(2))[0]
        freeblock_size = struct.unpack('>H', f.read(2))[0]
        info_offset = ("Offset: {} ({} Bytes) ".format(hex(offset +
first_free_block),int(offset + first_free_block)))
        info_fbsize = ("FB_Size: {} ({} Bytes) ".format(hex(freeblock_size),
int(freeblock_size)))
        info_eofb = ("eofb: {} ({} Bytes) ".format(hex(offset + first_free_block
+ freeblock_size), int(offset + first_free_block + freeblock_size)))
        info_nextfb = ("Next Freeblock: {} ({} Bytes)\n".format(hex(offset +
freeblock), int(offset + freeblock)))
        info_seek = ("Seek: {} ({} Bytes)".format(hex(offset + freeblock),
int(offset + freeblock)))
        info = info_offset + info_fbsize + info_eofb + info_nextfb
        output.write(str(info))
        content = f.read(freeblock_size - 4)
        output.write("freeblock_content_hex: {}\n\n".format(content))
        first_free_block = freeblock
    walframe += 1
    offset = offset + pagesize + 24

def sqlitefile(f, output, filesize):
    f.seek(16)
    pagesize = struct.unpack('>H', f.read(2))[0]
    offset = 0
    while offset < filesize:
        f.seek(offset)
        ident_page = struct.unpack('>B', f.read(1))[0]
        if ident_page == 13:
            first_free_block = struct.unpack('>H', f.read(2))[0]
            while first_free_block != 0x0:
                f.seek(offset + first_free_block)
                freeblock = struct.unpack('>H', f.read(2))[0]
                freeblock_size = struct.unpack('>H', f.read(2))[0]
                info_offset = ("Offset: {} ({} Bytes) ".format(hex(offset +
first_free_block),int(offset + first_free_block)))
                info_fbsize = ("FB_Size: {} ({} Bytes) ".format(hex(freeblock_size),
int(freeblock_size)))
                info_eofb = ("eofb: {} ({} Bytes) ".format(hex(offset + first_free_block
+ freeblock_size), int(offset + first_free_block + freeblock_size)))

```

```

        info_nextfb = ("Next Freeblock: {} ({} Bytes)\n".format(hex(offset +
freeblock), int(offset + freeblock)))
        info_seek = ("Seek: {} ({} Bytes)".format(hex(offset + freeblock),
int(offset + freeblock)))
        info = info_offset + info_fbsize + info_eofb + info_nextfb
        output.write(str(info))
        content = f.read(freeblock_size - 4)
        output.write("freeblock_content_hex: {}\n\n".format(content))
        first_free_block = freeblock

    offset = offset + pagesize

database = sys.argv[1] #'/home/blackhand/Desktop/notes_db-wal'
output_file = sys.argv[2] #'/home/blackhand/Desktop/test'

stats = os.stat(database)
filesize = stats.st_size

try:
    f = open(database,'rb')
except:
    print("File could not be open")

try:
    output = open(output_file, 'w')
except:
    print("Cannot open outputfile")

f.seek(0)
header = f.read(4)

if header == b'SQLi':
    sqlitefile(f, output, filesize)
elif header == b'7\x7f\x06\x82' or header == b'7\x7f\x06\x83':
    wal(f, output, filesize)
else:
    print("No valid SQLite-File or Write-Ahead-Log")

```

## 9.7 Wal2sqlite.py

```

import sqlite3, csv, os, sys

filename = sys.argv[1] # z.B. "/home/blackhand/Desktop/places.sqlite"
wallog_folder = sys.argv[2] # z.B.
"/home/blackhand/Desktop/wal_out/places.sqlite-wal/WALs/"
splitsym = "§"

```

```
def connection_to_db(db_name):
    conn = None
    try:
        with sqlite3.connect(db_name) as conn:
            cursor = conn.cursor()
            return cursor, conn

    except sqlite3.Error as e:
        print(e)

def get_files(dir):
    files = []
    for filenames in os.listdir(dir):
        files.append(os.path.join(dir, filenames))
    return files

def parse_wallog_file(filename):
    file_content = []
    with open(filename, 'r') as wallog:
        wallog = wallog.readlines()[1:-1]
        for line in wallog:
            temp = line[:-2]
            #print(temp)
            file_content.append(temp)
    return file_content

def parse_wallog_files(folder):
    temp_files = []
    temp_list = []
    temp_files = get_files(folder)
    for file in temp_files:
        with open(file, 'r') as wallog:
            wallog = wallog.readlines()[1:2]
            for line in wallog:
                temp_list.append(get_colcount_from_line(line))
    temp_list = check_length(temp_list)
    return temp_files, temp_list

def get_tables_from_db(f):
    tables = []
    conn = None
    try:
        with sqlite3.connect(f) as conn:
            cursor = conn.cursor()
            cursor.execute("Select name FROM sqlite_master WHERE
type='table';")
            table_names = cursor.fetchall()
            for table in table_names:
                table = str(table)[2:(len(table) - 4)]
```

```
        finding = table.find("_copy")
        if finding == -1 and table != "sqlite_stat1":
            tables.append(table)
    return tables

except sqlite3.Error as e:
    print(e)

finally:
    if conn:
        conn.close()

def get_col_from_db(f,tn):
    list1 = []
    list2 = []
    conn = None
    try:
        with sqlite3.connect(f) as conn:
            cursor = conn.cursor()
            for table in tn:
                cursor.execute("PRAGMA table_xinfo({})".format(table))
                list1.append(cursor.fetchall())

    except sqlite3.Error as e:
        print(e)

    finally:
        if conn:
            conn.close()
    return list1

def get_colcount_from_line(line):
    colcount = line.count(splitsym)
    return colcount

def check_length(items):
    temp_list = []
    for item in items:
        for db, length in db_tn_and_col_count.items():
            if item != length:
                continue
            else:
                temp_list.append(db)
                break
    return temp_list

def get_col_count_from_db_1(a):
    temp = []
    for items in a:
        temp.append(len(items))
```



```

return temp

def insert_logfile_content_to_db(db_filename, db_content, logfile_content):
    cursor, conn = connection_to_db(db_filename)
    for file, db in logfile_content.items():
        dbc = None
        count = None
        dbc = db + "_copy"
        create_copy_of_table(db_filename, db)
        get_items_from_log_file = parse_wallog_file(file)
        count = (get_items_from_log_file[0].count(splitsym)) + 1
        temp_placeholder = (count * "?," ) + "?"
        for item in get_items_from_log_file:
            item = item.split(splitsym)
            item = [0 if item == 'NULL' else item for item in item]
            item.append((file.split("/"))[-1:][0])
            try:
                cursor.execute("PRAGMA foreign_keys=off")
                cursor.execute('INSERT OR REPLACE INTO {} VALUES
({})'.format(dbc,temp_placeholder), item)
                conn.commit()
            except sqlite3.Error as e:
                print(e,item)

        finally:
            conn.commit()

def create_copy_of_table(db_filename, tn):
    tn_copy = None
    cursor, conn = connection_to_db(db_filename)
    tn_copy = tn + "_copy"
    cursor.execute("SELECT * FROM sqlite_master WHERE type='table' AND
name='{}' OR name='{}'".format(tn,tn_copy))
    content = cursor.fetchall()
    if len(content) == 1:
        c = content[0][4].replace(tn, tn_copy)
        c = c.replace("INTEGER PRIMARY KEY", "INTEGER")
        cursor.execute(c)
        c = ("ALTER TABLE {} ADD COLUMN log_filename
LONGVARCHAR".format(tn_copy))
        cursor.execute(c)
        conn.commit()
        return False
    else:
        return True

if __name__ == '__main__':
    tables_from_db = get_tables_from_db(filename)
    col_from_db = get_col_from_db(filename, tables_from_db)

```

```
db_tn_and_col_count = dict(zip(tables_from_db,  
get_col_count_from_db_1(col_from_db)))  
  
log_filename, likely_tables = parse_wallog_files(wallog_folder)  
lfn_and_ltn = dict(zip(log_filename, likely_tables))  
  
insert_logfile_content_to_db(filename, db_tn_and_col_count, lfn_and_ltn)
```

**Literaturverzeichnis**

- [1] „Most Widely Deployed SQL Database Engine,“ Hipp, Wyrick & Company, Inc, 8 Januar 2022. [Online]. Available: <https://www.sqlite.org/mostdeployed.html>. [Zugriff am 29 Juni 2024].
- [2] D. Bieber, „Facebook Messenger SQL Queries | David Bieber,“ [Online]. Available: <https://davidbieber.com/snippets/2020-04-12-fb-messenger-sql/>. [Zugriff am 29 Juni 2024].
- [3] „How To Decrypt WeChat EnMicroMsg.db Database? - Forensic Focus,“ Forensic Focus, 1 Oktober 2014. [Online]. Available: <https://www.forensicfocus.com/articles/decrypt-wechat-enmicromsgdb-database/>. [Zugriff am 29 Juni 2024].
- [4] D. Curry, „Messaging App Revenue and Usage Statistics (2024) - Business of Apps,“ Business of Apps, 8 Januar 2024. [Online]. Available: <https://www.businessofapps.com/data/messaging-app-market/>. [Zugriff am 29 Juni 2024].
- [5] Wikimedia Foundation, Inc, „SQLite - Wikipedia,“ Wikimedia Foundation, Inc, 2 Mai 2024. [Online]. Available: <https://en.wikipedia.org/wiki/SQLite>. [Zugriff am 29 Juni 2024].
- [6] Statista, „Internet and social media users in the world 2024 | Statista,“ Statista, 22 Mai 2024. [Online]. Available: <https://www.statista.com/statistics/617136/digital-population-worldwide/>. [Zugriff am 29 Juni 2024].
- [7] Statista, „Chart: Google's Chrome Has Taken Over the World | Statista,“ Statista, 1 September 2023. [Online]. Available: <https://www.statista.com/chart/30734/browser-market-share-by-region/>.

[Zugriff am 29 Juni 2024].

- [8] J. B. K. B. & S. L. Sangjun Jeon, „Personal and Ubiquitous Computing,“ Bd. 16, Nr. 6, pp. 708-715, 2012.
- [9] N. Z. M. X. Weixin SHU, „A History Records Recovering Method Based on WAL,“ *Journal of Computational Information Systems*, Bd. 10, Nr. 20, p. 8973–8982, 2014.
- [10] S. S. F. F. Sebastian Nemetz, „A standardized corpus for SQLite database forensics,“ *Digital Investigation*, Bd. 24, Nr. March, pp. 121-130, 2018.
- [11] H. B. Christian Meng, „bring2lite: A Structural Concept and Tool for Forensic Data Analysis,“ *Digital Investigation*, Bd. 29, Nr. July, pp. 31-41, 2019.
- [12] Z. Q. M. H. O. C. Eman Daraghmi, „Forensic Operations for Recognizing SQLite Content (FORC):An Automated Forensic Tool for Efficient SQLite Evidence,“ *applied sciences*, Bd. 13, Nr. 19, 2023.
- [13] Sanderson Forensics , „Forensic Toolkit for SQLite,“ Sanderson Forensics , 2024. [Online]. Available: <https://sqliteforensicstoolkit.com/sqlite-forensic-toolkit/>. [Zugriff am 29 06 2024].
- [14] Belkasoft, „SQLite Forensics with Belkasoft X,“ Belkasoft, 2024. [Online]. Available: <https://belkasoft.com/sqlite>. [Zugriff am 29 06 2024].
- [15] Magnet Forensics, „Enhanced SQLite Viewer Introduced in AXIOM 3.1,“ Magnet Forensics, 2024. [Online]. Available: <https://www.magnetforensics.com/resources/enhanced-sqlite-viewer-introduced-in-axiom-3-1/>. [Zugriff am 29 06 2024].
- [16] National Institute of Standards and Technology, „Software Quality Group,“ National Institute of Standards and Technology, 14 September 2023. [Online]. Available: <https://www.nist.gov/itl/ssd/software-quality-group/computer-forensics-tool-testing-program-cftt/cftt-technical/sqlite>.

- [Zugriff am 29 06 2024].
- [17] D. Pawlaszczyk, „FQLITE Forensic SQLite Data Recovery Tool,“ 2024. [Online]. Available: <https://www.staff.hs-mittweida.de/~pawlaszc/fqlite/>. [Zugriff am 29 06 2024].
- [18] alitrack, „GitHub - alitrack/undark: Undark - a SQLite recovery tool for deleted data or corrupt database,“ 2024. [Online]. Available: <https://github.com/alitrack/undark>. [Zugriff am 29 Juni 2024].
- [19] bring2lite, „GitHub - bring2lite/bring2lite,“ 2024. [Online]. Available: <https://github.com/bring2lite/bring2lite>. [Zugriff am 29 Juni 2024].
- [20] Packt Publishing, „Learning-Python-for-Forensics-Second-Edition/Chapter12/wal\_crawler.py at master · PacktPublishing/Learning-Python-for-Forensics-Second-Edition · GitHub,“ Packt Publishing, 2024. [Online]. Available: [https://github.com/PacktPublishing/Learning-Python-for-Forensics-Second-Edition/blob/master/Chapter12/wal\\_crawler.py](https://github.com/PacktPublishing/Learning-Python-for-Forensics-Second-Edition/blob/master/Chapter12/wal_crawler.py). [Zugriff am 29 Juni 2024].
- [21] E. Schicker, Datenbanken und SQL, Wiedbaden: Springer Vieweg, 2017.
- [22] P. Sanderson, SQLite Forensics, First Edition Hrsg., Cornwall: Amazon Fullfillment, 2018.
- [23] Hipp, Wyrick & Company, Inc., „SQLite Home Page,“ Hipp, Wyrick & Company, Inc., 15 April 2024. [Online]. Available: <https://www.sqlite.org/>. [Zugriff am 29 Juni 2024].
- [24] Mozilla, „gecko-dev/third\_party/sqlite3 at master · mozilla/gecko-dev · GitHub,“ 2024. [Online]. Available: [https://github.com/mozilla/gecko-dev/tree/master/third\\_party/sqlite3](https://github.com/mozilla/gecko-dev/tree/master/third_party/sqlite3). [Zugriff am 29 Juni 2024].
- [25] Hipp, Wyrick & Company, Inc., „SQLite Library Footprint,“ Hipp, Wyrick & Company, Inc., 30 Juli 2023. [Online]. Available:

- <https://www.sqlite.org/footprint.html>. [Zugriff am 29 Juni 2024].
- [26] Hipp, Wyrick & Company, Inc., „About SQLite,“ Hipp, Wyrick & Company, Inc., 10 Oktober 2023. [Online]. Available: <https://www.sqlite.org/about.html>. [Zugriff am 29 Juni 2024].
- [27] Hipp, Wyrick & Company, Inc., „Database File Format,“ Hipp, Wyrick & Company, Inc., 17 Juni 2024. [Online]. Available: [https://www.sqlite.org/fileformat2.html#b\\_tree\\_pages](https://www.sqlite.org/fileformat2.html#b_tree_pages). [Zugriff am 30 Juni 2024].
- [28] Wikimedia Foundation, Inc., „Variable-length quantity - Wikipedia,“ Wikimedia Foundation, Inc., 15 Mai 2024. [Online]. Available: [https://en.wikipedia.org/wiki/Variable-length\\_quantity](https://en.wikipedia.org/wiki/Variable-length_quantity). [Zugriff am 6 Juli 2024].
- [29] Hipp, Wyrick & Company, Inc., „Pragma statements supported by SQLite,“ Hipp, Wyrick & Company, Inc., 16 April 2024. [Online]. Available: [https://www.sqlite.org/pragma.html#pragma\\_secure\\_delete](https://www.sqlite.org/pragma.html#pragma_secure_delete). [Zugriff am 30 Juni 2024].
- [30] B. Carrier, „www.sleuthkit.org: About,“ BasisTeceh, 2023. [Online]. Available: <https://www.sleuthkit.org/about.php>. [Zugriff am 22 Juli 2024].
- [31] M. DeGrazia, „Another Forensics Blog: Python Parser to Recover Deleted SQLite Database Data,“ 6 November 2013. [Online]. Available: <http://az4n6.blogspot.com/2013/11/python-parser-to-recover-deleted-sqlite.html>. [Zugriff am 13 Juli 2024].
- [32] Magnet Forensics, „Enhanced SQLite Viewer Introduced in AXIOM 3.1 - Magnet Forensics,“ Magnet Forensics, 2024. [Online]. Available: <https://www.magnetforensics.com/resources/enhanced-sqlite-viewer-introduced-in-axiom-3-1/>. [Zugriff am 16 Juli 2024].



---

**Abbildungsverzeichnis**

Abbildung 3-1 Abbildung eines B-tree-Headers .....	20
Abbildung 3-2 Ausschnitt eines Datensatzes mit hexadezimaler Darstellung der ROWID .....	21
Abbildung 3-3 Schematischer Aufbau eines Datensatzes in einer B-tree- Blattseite [22, p. 30] .....	22
Abbildung 3-4 Beispieldatensatz einer Zeile in einer SQLite-Datenbank .....	24
Abbildung 5-1 Eigenschaften der Datenbank places.sqlite .....	43
Abbildung 5-2 Ausschnitt der Tabelle moz_places.....	44
Abbildung 5-3 Umrechnung des Datumformats.....	45
Abbildung 5-4 Artefakte in favicons.sqlite.....	46
Abbildung 5-5 Unallocated Space der Datenbank places.sqlite .....	47
Abbildung 5-6 Anzeige Web-Related unter Axiom .....	48
Abbildung 5-7 SQL-Abfrage der sqlite_master in Axiom .....	48
Abbildung 5-8 Create Table Statement der Tabelle moz_places .....	49
Abbildung 5-9 Abfrage zur Feststellung von Lücken bei ROWID der Tabelle moz_places .....	49
Abbildung 5-10 Tabelleninhalte der Datenbank places.sqlite in Autopsy .....	51
Abbildung 5-11 Manuelles Finden von fehlenden IDs in der Datenbank places.sqlite .....	52
Abbildung 5-12 Ausschnitt von Einträgen aus der Datenbank mittels FQLite .....	53
Abbildung 5-13 Ausschnitt von Einträgen aus dem WAL mittels FQLite .....	53
Abbildung 5-14 Gefundenes Artefakt der Webseite reddit.com mittels FQLite .....	53
Abbildung 5-15 Fehlermeldung bzgl. unstimmgiger Anzahl von Spalten des Pythonskripts .....	54



---

Abbildung 5-16 Ergebnis der Extraktion von Freeblocks im Szenario Firefox .....	54
Abbildung 5-17 Überblick der Tabellen nachdem Einträge aus dem WAL hinzugefügt wurden.....	55
Abbildung 5-18 Einträge die nicht in der Produktivdatenbank zu finden sind .....	55
Abbildung 5-19 Ausschnitt der fehlenden ROWIDs.....	56
Abbildung 6-1 Datenbankeigenschaften von notes_db in Belkasoft.....	58
Abbildung 6-2 Anzeige der ersten Zeilen der Tabelle entry.....	58
Abbildung 6-3 Letzter Eintrag in der Tabelle entry .....	58
Abbildung 6-4 Fehlender Eintrag in der Tabelle entry .....	59
Abbildung 6-5 Unterschiedliche ROWIDs für den gleichen Eintrag .....	59
Abbildung 6-6 Übersicht der verfügbaren Versionen des Eintrags Smalltalk ...	60
Abbildung 6-7 Ergebnis der SQL-Abfrage .....	62
Abbildung 6-8 Ergebnis der SQL-Abfrage um Erstellungszeitraum einzugrenzen .....	62
Abbildung 6-9 Ausschnitt der gefundenen Werte in einem Hexeditor .....	64
Abbildung 6-10 Anzeige einer Lücke in der Spalte _order .....	65
Abbildung 6-11 Auszug aus der Textansicht des WALs.....	65
Abbildung 6-12 gefilterte Einträge im WAL durch FQLite .....	66
Abbildung 6-13 Ergebnis der Extraktion von Freeblocks im Szenario Memorix .....	68

---

## Tabellenverzeichnis

Tabelle 3-1 Ausgewählte Header-Informationen einer SQLite-Datenbank- Datei [27].....	17
Tabelle 3-2 Aufbau eines B-tree Seitenheaders [27].....	19
Tabelle 3-3 Auflistung der Codierung für Datentypen [27].....	23
Tabelle 3-4 gekürzter Ausschnitt des Create Table-Statements der Tabelle Mitarbeiter .....	23
Tabelle 5-1 PRAGMA-Optionen der Datenbanken aus Firefox .....	56
Tabelle 6-1 Wiederhergestellte Versionen des Eintrags Smalltalk .....	60
Tabelle 6-2 Wiederhergestellter Inhalt der Notiz Smalltalk mittels Suche .....	63
Tabelle 6-3 Zusammenfassung der gefundenen Werte .....	64
Tabelle 6-4 Zeiträume der einzelnen Checkpoints .....	67
Tabelle 6-5 PRAGMA-Optionen der Datenbank notes_db .....	68
Tabelle 7-1 Bewertung der eingesetzten Software nach Bewertungskriterien .....	70

## Abkürzungsverzeichnis

ADB *Android Debug Bridge*  
BLOB *Binary Large Object*  
DBMS *Database Management System*

MSB *Most significant bit*  
Nonce *Number only used once*  
WAL *Write-Ahead-Log*

## **Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig, ohne unerlaubte fremde Hilfe und nur unter Verwendung der in der Arbeit aufgeführten Hilfsmittel angefertigt habe.

Ort, Datum

(Unterschrift)