Weiterentwicklung Forensischer Werkzeuge

Autoren:
Andreas Dewald
Felix Freiling
Sabine Seufert
Christian Zoubek

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Modul 1

Weiterentwicklung Forensischer Werkzeuge

Studienbrief 1: RAID

Studienbrief 2: Thumbcaches

Studienbrief 3: Blockhasher

Studienbrief 4: Threema

Autoren:

Andreas Dewald

Felix Freiling

Sabine Seufert

Christian Zoubek

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

© 2015 Felix Freiling
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (2. Dezember 2015)

Didaktische und redaktionelle Bearbeitung:

Das Werk einschließlich seiner Teile ist urheberrechtlich geschützt. Jede Verwendung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verfasser unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 160H11068 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

Einleitung zu den Studienbriefen **4**
I. Abkürzungen der Randsymbole und Farbkodierungen 4
II. Zu den Autoren 5
III. Modullehrziele 6

Einleitung **7**

Studienbrief 1 RAID **9**
1.1 Beschreibung 9
1.2 Aufgabenstellung 11
1.3 Lösungsvorschlag 12

Studienbrief 2 Thumbcaches **15**
2.1 Beschreibung 15
2.2 Aufgabenstellung 15
2.3 Lösungsvorschlag 16

Studienbrief 3 Blockhasher **17**
3.1 Beschreibung 17
3.2 Aufgabenstellung 17
3.3 Lösungsvorschlag 18

Studienbrief 4 Threema **21**
4.1 Beschreibung 21
4.2 Aufgabenstellung 21
4.3 Lösungsvorschlag 21

Verzeichnisse **23**
I. Abbildungen 23
II. Literatur 23

Index **24**

Einleitung zu den Studienbriefen

I. Abkürzungen der Randsymbole und Farbkodierungen

II. Zu den Autoren

Andreas Dewald

Felix Freiling

Sabine Seufert

Christian Zoubek

III. Modullehrziele

Dieses Modul soll auf spielerische Weise einen selbstbestimmten Umgang mit Werkzeugen digitaler Forensik lehren. Nach Abschluss dieses Moduls können Sie gängige Forensik-Werkzeuge mit neuer Funktionalität versehen. Außerdem lernen Sie, die Qualität von forensischer Software zu prüfen und zu bewerten.

Einleitung

Der zentrale Untersuchungsgegenstand der forensischen Informatik ist die *digitale Spur* [Dewald and Freiling, 2011], die in vielen unterschiedlichen Erscheinungsformen auftreten kann: als (gelöschte) Datei, als gesetztes Bit im Hauptspeicher eines Rechners, oder als Eintrag in einer Logdatei. Zwar sind digitale Spuren immer an materielle Objekte gebunden, aber die Zuordnung der Daten zu einem bestimmten Datenträger ist im Gegensatz zu physischen Spuren eher beliebig. Stets aber müssen die gespeicherten Daten ausgelesen und interpretiert werden, wozu nicht nur Sachverstand sondern auch eine Sammlung von Werkzeugen notwendig sind.

Werkzeuge für die Aquise und Analyse digitaler Spuren gibt es viele. Neben den kommerziellen Alleskönnern wie X-Ways, EnCase oder FTK gibt es mittlerweile auch eine große Menge von quelloffenen Werkzeugen wie TSK[1] oder Rekall[2], die digitale Spuren analysieren, interpretieren und darstellen können.

Obwohl diese Werkzeuge beständig fortentwickelt werden, so kann als gesichert gelten, dass sie noch viele Fehler enthalten [Wundram et al., 2013]. Es ist demnach wichtig, durch eigene Tests strukturiert bestimmte Funktionalitäten der Software testen zu können. Hierzu benötigt man Programmierkenntnisse, um etwa Werkzeuge automatisiert zu starten und mit Daten zu füttern.

Programmierkenntnisse sind im Kontext von forensischen Werkzeugen aber auch aus einem anderen Grund empfehlenswert, denn in der Praxis stoßen existierende Werkzeuge schnell an ihre Grenzen. Wenn etwa ein vorliegendes Dateisystem von meinem Forensik-Werkzeug nicht interpretiert werden kann, dann bin ich aufgeschmissen, es sei denn, ich besitze die Fähigkeit, das Werkzeug zu erweitern oder mir ein eigenes (kleines) Werkzeug für diesen Zweck zu schreiben.

Das Zertifikatsmodul

Die beiden eben formulierten Problemstellungen, Testen und Weiterentwicklung forensischer Werkzeuge, sind der Ausgangspunkt dieses Moduls. Wir wollen im Rahmen dieses Moduls den Teilnehmern Hilfe zur Selbsthilfe vermitteln. Sie sollen die Scheu und auch in gewisser Weise den Respekt vor ihren Werkzeugen verlieren: Sie sollen sie (durch umfangreiche Tests) fordern und (durch gezielte Weiterentwicklung) fördern.

Dieses Zertifikatsmodul ist auf eine Bearbeitungszeit von 2 Monaten ausgelegt. Die Zielgruppe sind Personen, die schon erste Erfahrungen im Bereich digitaler Forensik gesammelt und keine Scheu vor dem Programmieren haben. Sinnvoll sind Programmierkenntnisse in einer Skriptsprache wie Python, sowie Kenntnisse im Umgang mit der Kommandozeile. In diesem Modul sollen beispielhaft kleinere Projekte im Kontext forensischer Werkzeuge durchgeführt werden. Die Aufgabenstellungen können dabei durchaus selbst bestimmt werden und aus dem eigenen Arbeitskontext stammen.

Die Materialien der folgenden Studienbriefe dokumentieren exemplarisch einige solche Aufgaben. Weitere Aufgaben können beispielsweise aus dem Kontext des DFXML-Projekts [Garfinkel, 2012] stammen, indem man existierende Werkzeuge mit XML-Ausgaben versieht (oder entsprechende Filter schreibt), um sie interoperabel zu machen. Hilfreiche Bibliotheken dazu findet man beispielsweise in Form

[1] http://www.sleuthkit.org/
[2] http://www.rekall-forensic.com/

des Parsergenerators PLY[3] oder anderer Werkzeuge im Kontext von Python und XML[4]. Teilnehmer sollen ihr Werkzeug systematisch testen und ihr Werkzeg abschließend gegenüber anderen Teilnehmern präsentieren. Gute Werkzeuge sollen über das Internet anderen Praktikern zur Verfügung gestellt werden.

Danksagungen

Die Inspiration zu diesem Modul kam aus zwei Quellen. Zum einen stammte sie aus der Lektüre des Buches "Computer Forensik Hacks" von Lorenz Kuhlee und Victor Völzow [Kuhlee and Völzow, 2012], einem intelligenten und hilfreichen Praxisbuch der digitalen Forensik, das mit vielen kleinen Kniffen und Hinweisen bewußtseinserweiternd wirkt in der Welt der forensischen Werkzeuge. Zum anderen inspirierend wirkte die Lehrveranstaltung "Forensic Hacks", die Andreas Dewald seit 2014 regelmäßig an der Friedrich-Alexander-Universität im Präsenzstudium Informatik anbietet, und die immer wieder zeigt, wie einfach und schön Lösungen zu vermeintlich schwierigen Analyseproblemen aussehen können.

Das vorliegende Textmaterial entstand im Rahmen des vom BMBF geförderten Projekts "Open C3S".

Allen Teilnehmern wünschen wir viel Spaß und viel Erfolg!

[3] http://www.dabeaz.com/ply/
[4] http://pyxml.sourceforge.net/topics/

Studienbrief 1 RAID

1.1 Beschreibung

Ein *RAID*-System ist ein Verbund mehrerer physischer Festplatten zu einem zusammenhängenden logischen Laufwerk („Redundant Array of Independent Disks“). Dieser Verbund enthält in den meisten Fällen redundante Informationen, um Datenverlust zu verhindern und Ausfallsicherheit zu gewährleisten. Abhängig vom verwendeten RAID-Typ ist es auch möglich, die Lese- und Schreibleistung der Laufwerkzugriffe zu erhöhen. Durch die inzwischen geringen Anschaffungskosten von Festplatten bieten sich RAID-Systeme mit den genannten Vorteilen besonders für den Betrieb in Serverumgebungen an, sind aber auch im privaten Gebrauch anwendbar – etwa als *NAS*-System („Network Attached Storage“).

Der Zugriff auf das logische Laufwerk wird von einem sogenannten *RAID-Controller* transparent für Betriebs- und Dateisysteme durchgeführt. Da die Daten des Laufwerks – je nach RAID-Typ – nicht in ihrer logischen Reihenfolge auf den Festplatten anzutreffen sind, ist eine forensische Post-Mortem-Analyse nicht ohne Weiteres möglich. Im Kontext der forensischen Informatik sind Szenarien, bei denen Festplatten zerstört oder beschädigt wurden oder unauffindbar sind, ein Problem. Durch die bereits erwähnte Datenredundanz können derartige Systeme unter Umständen wiederhergestellt werden. Um eine solche Wiederherstellung zu ermöglichen, ist es notwendig, den RAID-Typ und ggf. die Parameter des verwendeten RAID-Controller zu ermitteln. Dies gestaltet sich, je nach RAID-Typ, unterschiedlich komplex.

Die meisten RAID-Typen verteilen die zu speichernden Daten blockweise über mehrere Festplatten. Dies wird *Striping* genannt. Dem Raid-Controller müssen für das Striping zwei Parameter bekannt sein, *Stripe-Size* und *Stripe-Map*. Die Stripe-Size (auch *Chunk-Size* genannt) beschreibt welche Größe ein zusammenhängender Block hat und kann in den meisten Fällen dem RAID-Controller beim Erstellen des RAID-Systems übergeben werden. Der RAID-Controller definiert initial die Stripe-Map, welche die Verteilung der Speicheradressen über die angeschlossenen Speichermedien beschreibt. Die Stripe-Map ist i.d.R. zyklisch. Ein einfaches Beispiel für ein Striping stellt RAID-0 dar. Bei diesem RAID-Typ werden die Blöcke linearisiert über die Festplatten geschrieben, wie in Abb. 1.1 zu sehen ist.

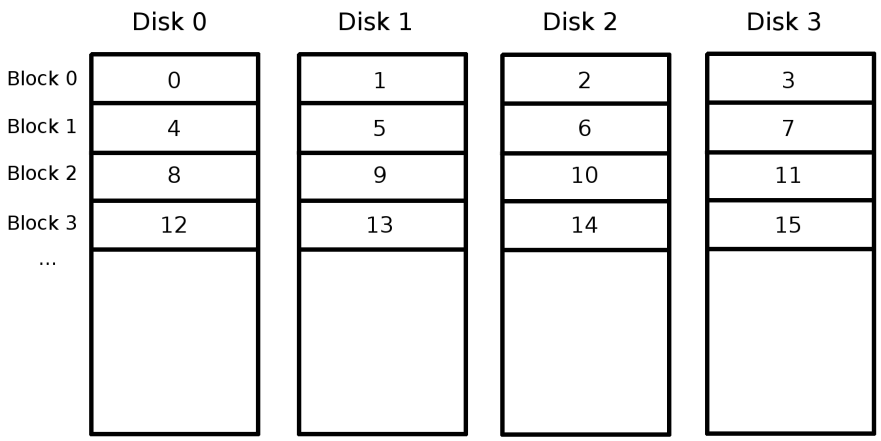


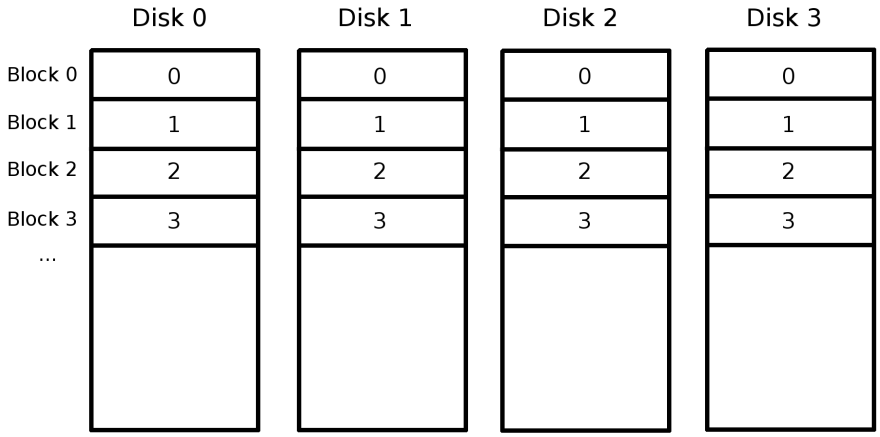
Abb. 1.1: Beispielverteilung für ein RAID-0-System mit vier Festplatten.

Da aufeinanderfolgende Blöcke sich so auf n unterschiedlichen Festplatten mit eigenen Schreib-/Leseköpfen befinden, erhöht sich die Lese- und Schreibgeschwin-

digkeit im Idealfall jeweils um den Faktor n. Der gesamte verfügbare Speicher wird ausschließlich für Daten und ohne Abspeichern redundanter Informationen verwendet. Im Falle eines Festplatten-Ausfalls ist die Wiederherstellung der Daten weitestgehend unmöglich.

Im Kontrast zu RAID-0 spielt bei RAID-1 die Performance eine untergeordnete Rolle. Das Hauptaugenmerk liegt auf Ausfallsicherheit, welche durch Redundanz erzielt wird. Hierbei wird das logische Laufwerk, wie in Abb. 1.2 zu sehen, schlichtweg auf alle n Festplatten gespiegelt. Daraus folgt eine Absicherung gegen Datenverlust, selbst wenn $n-1$ Festplatten defekt sind. Allerdings wird so kein Effizienzgewinn beim Schreiben erzielt; vielmehr wird bei einem heterogenen Aufbau die langsamste Festplatte zum Flaschenhals. Die Lese-Geschwindigkeit kann enorm gesteigert werden, da die Festplatten unabhängig voneinander von jeweils anderen Sektoren lesen können. Darüber hinaus ist lediglich ein n-tel des verfügbaren Speicherplatzes nutzbar.

Abb. 1.2: Beispielverteilung für ein RAID-1-System mit vier Festplatten.



Alle anderen RAID-Typen versuchen die Vorteile von RAID-0 und RAID-1 zu kombinieren und die Nachteile zu minimieren und bedienen sich der Redundanz über Paritätsbits. Als Parität eines Binärtextes wird die Information darüber bezeichnet, ob die Summe ihrer Bits ungerade ist. RAID-2 benutzt zur Fehlerkorrektur Hamming-Codes und bildet die zu speichernden Bitfolgen auf größere Folgen mit zusätzlichen Paritätsbits auf separaten Datenträgern ab. RAID-2 findet in der Praxis heute kaum noch Verwendung. Bei RAID-3 werden die Daten byteweise durch Striping über die Festplatten verteilt, wobei eine der Platten als Paritätsplatte dient. Darunter leidet die Schreib-Geschwindigkeit, da die Paritätsplatte bei jeder Operation angesprochen werden muss und so den Flaschenhals des gesamten RAID-Systems darstellt. RAID-4 ist eine Weiterentwicklung von RAID-3 und verteilt die Daten nicht byteweise, sondern blockweise. Der Nachteil einer dedizierten Paritätsplatte bleibt auch bei RAID-4 bestehen. RAID-5 ist, zusammen mit RAID-0 und RAID-1, einer der am häufigsten in der Praxis anzutreffenden RAID-Typen. Hierbei wird die Parität jedoch nicht auf einer dedizierten Festplatte, sondern über sämtliche Festplatten verteilt gespeichert. Eine Möglichkeit, die Paritätsblöcke in einem RAID-5-System zu verteilen, ist in Abb. 1.3 zu sehen.

Somit kombiniert RAID-5 Redundanz durch Paritätsinformationen und Zugriffseffizienz durch Striping. Die Speicherplatzeffizienz ist hierbei höher als bei RAID-1, da nur ein n-tel des physisch verfügbaren Speicherplatzes als Paritätsinformation genutzt wird. Da die Erhebung der Parität über ein einziges Bit eine Kopie ist, benötigt ein RAID-5-System mindestens drei Festplatten. Das Vorhandensein von

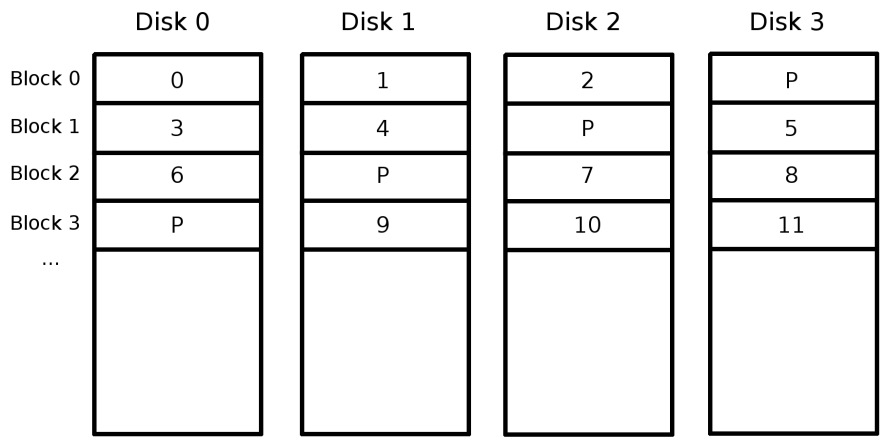


Abb. 1.3: Beispielverteilung für ein RAID-5-System mit vier Festplatten.

Partitätsinformationen erlaubt die Wiederherstellung aller Daten, wenn maximal eine Festplatte ausfällt. Dadurch, dass effektiv eine Festplatte keine Nutzdaten enthält, sind die Zugriffszeiten auf den Speicher idealerweise um den Faktor $n-1$ erhöht.

1.2 Aufgabenstellung

Um ein RAID-System wiederherzustellen, benötigt man (neben den oben genannten Punkten) Kenntnis des RAID-Typs, der Stripe-Size und ggf. der Stripe-Map. Die logische Reihenfolge der Festplatten ist hierbei i.d.R. nicht bekannt. Wie bereits erwähnt, ist es durchaus möglich, dass Festplatten fehlen oder beschädigt worden sind.

Programmieren Sie eine Software, die für eine gegebene Menge von Festplatten eines RAID-Verbunds imstande ist, die Parameter und den RAID-Typ zu ermitteln und seine Daten als ein zusammenhängendes, einhängbares Image auszugeben. Es soll möglich sein, anzugeben, aus wievielen Festplatten der analysierte Verbund ursprünglich bestand. Für RAID-Typen mit Redundanz soll Ihr Programm in der Lage sein, bei fehlenden Festplatten – wenn möglich – eine Wiederherstellung vorzunehmen. Falls Paritätsinformationen zur Verfügung stehen und alle Festplatten vorhanden waren, soll Ihr Programm mit diesen überprüfen, ob Ihr Ergebnis gültig ist. Ihnen werden verschiedene Testfälle der RAID-Typen 0, 1 und 5 zur Verfügung gestellt.

Gehen Sie folgendermaßen vor:

- Ermitteln Sie den RAID-Typ. Alternativ können Sie es dem Nutzer überlassen, diesen zu erraten.
- Stellen Sie fehlende und ggf. defekte Festplatten abhängig vom RAID-Typ fest und stellen Sie diese wieder her. Ihr Programm soll dies melden und abbrechen, falls eine Wiederherstellung unmöglich ist.
- Ermitteln Sie die Stripe-Size.
- Prüfen Sie blockweise, welche der Festplatten Paritätsinformationen beinhaltet. Diese folgt i.d.R. keinem bestimmten Muster wie etwa Text- oder Bilddateien. Beachten Sie hierbei, dass dies bei komprimierten oder verschlüsselten Dateien u.U. nicht zutrifft.

- Ermitteln Sie die Stripe-Map und ihren Zyklus.
- Ermitteln Sie die logische Reihenfolge der Festplatten. Machen Sie sich zunutze, dass die erste Festplatte die Partitionstabelle des Laufwerks enthalten muss. Falls Sie eine Kopie dieser Tabelle finden, muss eine der beiden die Paritätsinformation und alle anderen Festplatten ausgenullt sein.
- Beachten Sie, dass das RAID-System nicht an der physischen Adresse 0 anfangen muss. In diesem Fall müssen Sie mit einem Offset arbeiten, den Sie zuvor ermitteln müssen.
- Fügen Sie nun, wie oben beschrieben, die Daten zu einem zusammenhängenden Image zusammen und verifizieren Sie Ihr Ergebnis mittels Hashsummen.
- Da RAID-Systeme in der Praxis sehr groß werden können, sollten Sie die Ausführungszeit Ihres Programms bedenken.
- Führen Sie anschließend eine forensische Analyse auf dem so erstellen Image durch. Ihnen wird ein Testfall für diesen Zweck zur Verfügung gestellt.

1.3 Lösungsvorschlag

Erwägen Sie die folgenden Vorschläge bei der Konzeption und Implementierung Ihres Programms:

Ermittlung des RAID-Typs

Mittels einer bitweisen XOR-Operation über einige Blöcke können Sie den RAID-Typ i.d.R. schnell ermitteln. Wählen Sie hierzu Blöcke aus, die Informationen $\neq 0$ enthalten. Ergibt diese Operation immer 0, sind die Blöcke identisch und damit wahrscheinlich Teil eines RAID-1-Systems. Ergibt die Operation von $n-1$ Blöcken den übrigen Block, deutet dies auf die Existenz von Paritätsinformationen eines RAID-5-Systems hin. Ist das Ergebnis für verschiedene Blöcke frei von Mustern, liegt vermutlich ein RAID-0-System vor, weil dort keine Aussagen über XOR-Zusammenhänge getroffen werden können.

Bedenken Sie allerdings, dass Festplatten fehlen können. In diesem Fall sollten Sie heuristisch vorgehen, da sie über inexistente Daten keine allgemeingültigen Aussagen treffen können.

Ermittlung der Stripe-Size

Zur Ermittlung der Stripe-Size ist es unerlässlich, die Daten auf den Festplatten auf plötzliche Musterunterschiede zu untersuchen. Beispielsweise könnten Sie Texte suchen: Definieren Sie ein Wörterbuch und suchen Sie byteweise nach Wortfragmenten, auf die kein passendes Ende folgt. Passt der Inhalt eines anderen Blockanfangs zum Vervollständigen dieses Worts, haben Sie womöglich einen Teil der Stripe-Map-Definition und einen ersten Hinweis auf die Stripe-Size gefunden.

Ein anderer Musterunterschied, nach dem Sie suchen können, sind plötzliche Sprünge der Entropie in einem Block. Innerhalb von Dateien ändert sich die Entropie nur selten sprunghaft. Der Abstand zwischen zwei Entropiesprüngen könnte einen Hinweis auf die Stripe-Size liefern. Machen Sie sich zunutze, dass 512 Byte die kleinste Hardware-Blockgröße ist, die heutige Magnetfestplatten unterstützen.

Ermittlung der Stripe-Map

Wie bereits angedeutet, werden Sie zum Finden von Blockfolgen darauf zurückgreifen müssen, Blockanfänge und Blockenden semantisch miteinander zu verknüpfen. Das obige Beispiel beinhaltet das Verknüpfen von Wortteilen; Sie können aber auch mit der entropiebasierten Methode vorgehen. Hierbei würden Sie nach Blockfolgen suchen, bei denen sich die Entropie möglichst wenig ändert. Dabei spielen Nullblöcke eine große Rolle: Gehen Sie davon aus, dass Dateien keine langen Nullfolgen enthalten. Sie werden feststellen, dass Nullfolgen sich vor oder nach Dateien befinden. Beachten Sie dies bei der Ermittlung der Entropie eines Blocks, der bspw. zur Hälfte mit Nullen gefüllt ist. Überlegen Sie sich, warum Paritätsinformationen mit an Sicherheit grenzender Wahrscheinlichkeit nicht die Entropie 0 haben kann, sondern mit hoher Wahrscheinlichkeit die höchste Entropie im Gegensatz zu den Datenblöcken hat.

Studienbrief 2 Thumbcaches

2.1 Beschreibung

In früheren Windows-Versionen (Windows 2000 bis Windows XP) wird in jedem Verzeichnis eine Datei namens `thumbs.db` beim Betrachten von Bildern angelegt. Diese ist eine Datenbank aus *Thumbnails* (Vorschaubilder) zur einfacheren Navigation in Sammlungen von Dateien mit sichtbarem Inhalt oder für Icons im Allgemeinen. Durch diese vorgehaltenen Dateien verringert sich der zum Navigieren notwendige Datenfluss. Für forensische Untersuchungen ist hierbei interessant, dass die Thumbnails gelöschter Bilder nicht mitgelöscht werden.

Neuere Windows-Versionen (ab Windows Vista) speichern ihre Thumbnails stattdessen in zentralen Dateien pro Benutzer (*Thumb-Caches*). Diese befinden sich in `\Users\%username%\AppData\Local\Microsoft\Windows\Explorer\thumbcache_xx.db`. Innerhalb dieser Dateien lassen sich die Bilddateien unverschlüsselt vorfinden und können somit gecarvt oder mit entsprechenden Tools (z.B. `thumbcache-viewer`) geöffnet werden. Der Rückschluss von einem extrahierten Thumbnail auf das Originalbild ist nicht möglich, da die Thumb-Caches nur ID-Codes enthalten, die ohne das Vorhandensein des Originalbildes nichts aussagen. Da das Thumbnail allein für die forensische Analyse keine Aussagekraft hat, ist genau dieser Rückschluss entscheidend. Die Verbindung der Thumbnails aus den Thumb-Caches mit den Originaldateien kann bspw. durch die *Windows-Indexer*-Datenbank (`Windows.edb`, im *Extensible-Storage-Engine-Format* (ESEDB)) hergestellt werden, die mithilfe der `libesedb` interpretiert werden kann. Unter Anderem finden sich in dieser viele Metadaten der Suchindizierung. Sie befindet sich unter `C:\ProgramData\Microsoft\Search\Data\Applications\Windows\Windows.edb`.

In der dort angelegten Tabelle `SystemIndex_0A` befinden sich neben der Spalte 213 (`System_ThumbnailCacheId`) – wo die aus den Thumb-Caches entnommene ID vorzufinden ist – normalerweise noch weitere Metadaten, die für eine forensische Analyse von Interesse sind. Wurde das Bild nie vom Betriebssystem indiziert, gibt es keinen Eintrag in dieser Datenbank. Wurde das Bild nie betrachtet, existiert kein Thumbnail.

2.2 Aufgabenstellung

Programmieren Sie ein Werkzeug, welches Informationen aus den Thumb-Caches und der Windows-Indexer-Datenbank möglichst vollständig extrahiert. Ihre Ausgabe sollte in Form einer XML-Datei vorliegen, die Bildernamen und Metadaten enthält. Überlegen Sie sich hierfür ein sinnvolles Datenschema. Zusätzlich soll Ihr Programm die Ausgabe als LaTeX-Dokument beherrschen. Für die LaTeX-Ausgabe können Sie Ihre XML-Ausgabe mittels XSLT automatisiert konvertieren. Achten Sie darauf, dass Ihre Ausgabe mittels `\input{filename}` in Ihren forensischen Bericht eingefügt werden kann. In diesem sollen die gefundenen Thumbnails jeweils mit ihren Metadatentabellen dargestellt werden. Die Funde müssen gut dokumentiert werden; schließlich handelt es sich um ein forensisches Werkzeug.

Als optionale Bonusaufgaben können Sie Programme zum Öffnen der Thumb-Caches und der Windows-Indexer-Datenbank implementieren, anstatt auf fertige Viewer oder Bibliotheken zurückzugreifen.

Ihnen wird ein Windows-7-Image zur Untersuchung zur Verfügung gestellt.

2.3 Lösungsvorschlag

Lesen Sie sich in die Dokumentationen der beiden vorgeschlagenen Tools ein und überlegen Sie sich, wie Sie deren Ausgaben sinnvoll gegeneinander abgleichen können. Falls für ein Thumbnail keine Metadaten vorhanden sind, gehören diese trotzdem zu den Funden und müssen dokumentiert werden. Ein Beispiel für ein mögliches XML-Datenschema ist im folgenden Beispieldokument zu sehen:

```
<db>
<thumbnails user="admin">
  <thumbnail id="3141">
    <localFileName> ./extractedThumbnails/picture.jpg </localFileName>
    <metadata>
      <fileSize unit="Byte"> 64000 </fileSize>
      <checksums>
        <md5> ... </md5>
        <sha256> ... </sha256>
      </checksums>
      <exif>
        <resolution x="1376" y="768"/>
        ...
      </exif>
    </metadata>
  </thumbnail>

  <thumbnail id="2143">
  ...
  </thumbnail>
</thumbnails>
<thumbnails user="mrX">
...
</thumbnails>
</db>
```

Studienbrief 3 Blockhasher

3.1 Beschreibung

Ein Laufwerk mit fragmentierten Dateien, dessen Dateisystem-Metadaten zerstört wurden, lässt sich nicht mit einem Carver rekonstruieren. Dies trifft auch auf beschädigte Festplatten oder gelöschte Dateien zu. Für den Fall jedoch, dass auf einem solchen Laufwerk eine bestimmte, bereits bekannte Datei gesucht werden muss, bietet sich die Nutzung eines *Blockhashers* an. Grundsätzlich bildet er Hashsummen über die Blöcke der Festplatte und der zu suchenden Datei, um mögliche Spuren besagter Datei auf dem Laufwerk zu finden. Er besteht im Wesentlichen aus drei Komponenten:

- Der eigentliche Blockhasher teilt eine Datei in Blöcke einer konfigurierbaren Größe auf, bildet über diese die Hashsummen und speichert das Ergebnis als Liste ab.
- Die *Blockhash-Datenbank* ruft für alle Dateien eines Verzeichnisses den Blockhasher auf und führt die Ergebnislisten zu einer Datenbank zusammen.
- Der *Blockhash-Matcher* nimmt ein Festplattenimage und die Ausgabe der Blockhash-Datenbank an. Er wendet den Blockhasher auf das Image an und sucht nach Hashwert-Übereinstimmungen zwischen Image und der Datenbank.

Die Ergebnisse des Blockhash-Matchers müssen von Hand interpretiert werden: Das Vorhandensein von Blöcken mit gleichen Hashwerten wie Fragmente der gesuchten Datei alleine muss keinen Beweis darstellen, dass die Datei sich auf dem Laufwerk befunden hat. Die Bedeutung der Lage der gefundenen Blöcke muss gründlich erwogen werden. Insbesondere dann, wenn zusammenhängende Blockfolgen auf dem Laufwerk identisch mit Folgen der Originaldatei sind, besteht ein begründeter Verdacht darauf, dass die gesuchte Datei sich auf dem Datenträger befunden hat.

3.2 Aufgabenstellung

Implementieren Sie das oben vorgestellte, dreiteilige Programm. Ihr Blockhasher soll hierbei die MD5- und SHA256-Hashsummenbildung unterstützen und mit beliebigen Blockgrößen arbeiten können. Seine Ausgabe soll eine XML-Datei sein, in der sich die Metadaten der analysierten Datei und eine Liste der Blockadressen mit ihren Hashsummen befinden. Zusätzlich soll konfigurierbar sein, wieviele aufeinanderfolgende Blöcke erzeugt werden sollen und, ab welchem Offset damit begonnen werden soll, falls Teile einer Datei von Belang sind.

Ihre Blockhash-Datenbank soll folglich XML-Dateien des von Ihnen gewählten Schemas akzeptieren und diese in eine XML-Datei zusammenzufügen, die dann als Eingabedatei für Ihren Blockhash-Matcher dient.

Wie oben erwähnt, soll Ihr Blockhash-Matcher Ihren Blockhasher aufrufen, um eine Liste von Blöcken und Hashsummen des Images (oder eines Intervalls auf dem Image) zu erzeugen, mit der die gesuchten Dateien verglichen werden können. Mittels eines Parameters sollen Sie angeben können, ob Dateien rekonstruiert werden sollen, falls alle Blöcke auf diese Weise gefunden wurden. In diesem Fall soll ein Ordner angelegt werden, in den der Blockhash-Matcher alle rekonstruierten Dateien einfügt. Bei einer solchen Rekonstruktion soll er, zusätzlich zum Überprüfen der einzelnen Block-Hashsummen auch die Hashsummen über die gesamte Datei überprüfen.

Als Ausgabe (neben den vollständig rekonstruierten Dateien) soll Ihr Blockhash-Matcher eine XML-Datei ausgeben, die die Metadaten der Datenbankdatei, des Images und Informationen über alle Hashsummen-Übereinstimmungen enthält. Zu den Metadaten der Datenbankdatei gehören der Dateiname, die Größe und Hashsummen. Bei den Übereinstimmungen soll folgendes dokumentiert werden:

- Wieviele Blöcke einer Datei gefunden wurden (in absoluten und prozentualen Zahlen)
- Ob und wie häufig Blöcke mehrfach gefunden wurden (Überlegen Sie sich, was Mehrfachfunde bedeuten können.)
- Wie oft Blockfolgen auftraten, wie lang diese waren und wo sie sich befanden (Dokumentieren Sie redundante Blockfolgen und überlegen Sie sich, was die unterschiedlichen Längen der Folgen bedeuten.)
- Wenn Dateien vollständig wiederhergestellt wurden: Hashsummenvergleich der Wiederhergestellten mit der gesuchten Datei

Wenden Sie Ihr Werkzeug, Sleuthkit-Tools und Foremost auf den gegebenen Fall an und vergleichen Sie die Ergebnisse. Dokumentieren Sie diese in Tabellenform in einem Kurzbericht. Optional steht es Ihnen zu diesem Zweck frei, Ihre XML-Ausgabe (etwa mittels XSLT) in eine LaTeX-Datei zu konvertieren.

Wie gewohnt programmieren Sie ein forensisches Werkzeug. Dokumentieren Sie alle Arbeitsschritte und stellen Sie die Integrität des Beweismaterials sicher. Überlegen Sie, wie wahrscheinlich es ist, dass Sie Dateien fälschlicherweise nachweisen, die sich nie auf dem Datenträger befunden haben. Was bedeutet es, wenn verschiedene Dateien sich in Blockhashes teilweise gleichen? Bedeuten mehrfach auftretende Blöcke mehrere Kopien der gleichen Datei? Welche Folgen für die Beweiskraft hat es, wenn nur ein Teil aller Blöcke einer Datei gefunden werden (bspw. 90%)? Wie interpretieren Sie Anzahl und Abstand von Teilblockfolgen? Was bedeutet es, wenn manche Teilblöcke einer Datei öfter gefunden werden als andere?

3.3 Lösungsvorschlag

Bei der Implementierung des Blockhashers müssen Sie beachten, dass Sie mit teils sehr großen Images arbeiten. Erfahrungsgemäß können naive Implementierungen in ihrer Laufzeit um Faktoren von bis zu 60 variieren. Überlegen Sie sich folglich, wie Sie mittels Asynchronität, Multithreading und insbesondere Puffergrößen umgehen, um möglichst das Lesen vom Datenträger zu optimieren. Abhängig vom von Ihnen gewählten Zusammenspiel aus Blockhasher und Blockhash-Matcher, kann diese Überlegung für Sie auch für den Matcher von Belang sein.

Falls Sie sich darüber unsicher sind, welche Blockgröße sie für die Analyse wählen sollen, empfehlen wir mindestens 512-Byte-Blöcke. Zu große Blöcke sorgen dafür, dass der Block nicht ausschließlich durch die gesuchte Datei gefüllt wird, was bei gleichem Nutzinhalt zu unterschiedlichen Hashsummen führt. Bei 512-Byte-Blöcken – dem kleinsten Hardware-Block – können Sie davon ausgehen, dass der Blockrest beim Beschreiben mit Nullen aufgefüllt wird. Alternativ müssen Sie mit Verschnittblöcken umgehen, können dann aber für große Images größere Blöcke nutzen und diese schneller vergleichen. Überlegen Sie sich, woran Sie potenzielle letzte Blöcke einer Datei erkennen und, wie Sie diese mit der gesuchten Datei vergleichbar machen.

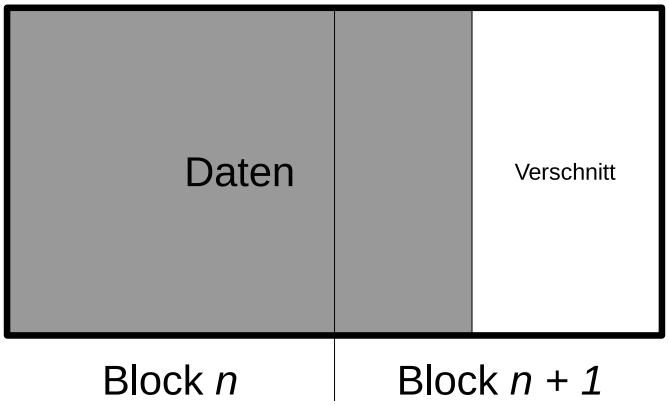


Abb. 3.1: Skizze eines Beispiels mit Verschnittblock

Studienbrief 4 Threema

4.1 Beschreibung

Threema ist ein (kostenpflichtiger) Smartphone-Messenger, der von einer Schweizer Firma als Alternative zu WhatsApp entwickelt wurde. Durch die Ende-zu-Ende-Verschlüsselung bei der Kommunikation, die Threema verwendet, kann kein zentraler Server die Unterhaltungen abhören. Threema-Nachrichten sind also auf dem Weg zum Empfänger verschlüsselt, werden aber auch lokal in einer *sqlite*-Datenbank gespeichert. Wie diese lokal verschlüsselt ist, hängt vom Betriebssystem ab. Bei iOS-Systemen hängt die Verschlüsselung vom Geräte-PIN ab und kann bei neueren Modellen durch Hardware-Maßnahmen verstärkt werden.

Auf Android-Systemen besitzt Threema eine eigene Verschlüsselungsmethode, die auf AES-256 basiert. Der hierfür verwendete Schlüssel wird beim ersten Start der App zufällig generiert und kann optional durch Setzen einer Passphrase erweitert werden. Die zusätzliche PIN der App sperrt (auf beiden Systemen) nur den Zugriff, wird aber nicht zur Verschlüsselung verwendet.

Wenn das Gerät entsperrt, kein PIN oder Passwort gesetzt ist oder, wenn PIN oder Passwort erraten werden können, ist somit der Zugriff auf eine Datenbank aller mit diesem Smartphone über Threema getätigten Unterhaltungen möglich.

4.2 Aufgabenstellung

Implementieren Sie ein Werkzeug zur forensischen Analyse einer bereits entschlüsselten Threema-Datenbank. Gehen Sie hierbei in zwei Schritten vor: Zunächst extrahieren Sie die Daten aus der *sqlite*-Datenbank und schreiben sie in eine XML-Datei. Als Nächstes nutzen Sie XSLT, um XPath-Anfragen auf dieser Datei abzusetzen und daraus eine LaTeX- und/oder HTML-Datei mit einer gut lesbaren Darstellung der Gesprächsverläufe samt der beteiligten Kontakte zu erzeugen.

Wie gewohnt schreiben Sie ein forensisches Werkzeug: Verwerten und dokumentieren Sie alle Informationen, die Sie erheben. Die Darstellung Ihrer Ergebnisse soll für Laien verständlich sein. Optional können Sie auch die Threema-Datenbank Ihres eigenen Smartphones für die Analyse verwenden.

4.3 Lösungsvorschlag

Wir empfehlen die Nutzung von *sqlite* als Programmierbibliothek für Ihr Programm. Es könnte von Vorteil sein, sich die Anfragesprache SQL und die Metadatenorganisation von *sqlite*-Datenbanken zu vergegenwärtigen. Zu diesem Zweck können Sie *sqlite* als interaktives Programm nutzen. Hier ein Beispiel für die XML-Ausgabe Ihres Programms:

```
<db>
  <table name="tablename">
    <row id="3141">
      <sender> Max Mustermann </sender>
      <text> Hallo </text>
      ...
    </row>
  </table>
  <table name="other_table">
  ...
  </table>
</db>
```

Verzeichnisse

I. Abbildungen

Abb. 1.1: Beispiel: RAID-0 9
Abb. 1.2: Beispiel: RAID-1 10
Abb. 1.3: Beispiel: RAID-5 11
Abb. 3.1: Beispiel: Blockverschnitt 19

II. Literatur

Andreas Dewald and Felix C. Freiling. *Forensische Informatik*. Books on Demand GmbH, 2011. ISBN 978-3-8423-7947-3. URL http://www.bod.de/index.php?id=296&objk_id=572434.

Simson Garfinkel. Digital forensics XML and the DFXML toolset. *Digital Investigation*, 8(3-4):161–174, 2012. doi: 10.1016/j.diin.2011.11.002. URL http://dx.doi.org/10.1016/j.diin.2011.11.002.

Lorenz Kuhlee and Victor Völzow. *Computer Forensik Hacks*. O'Reilly, 2012.

Martin Wundram, Felix C. Freiling, and Christian Moch. Anti-forensics: The next step in digital forensics tool testing. In *Seventh International Conference on IT Security Incident Management and IT Forensics, IMF 2013, Nuremberg, Germany, March 12-14, 2013*, pages 83–97, 2013. doi: 10.1109/IMF.2013.17. URL http://doi.ieeecomputersociety.org/10.1109/IMF.2013.17.

Index

Blockhash-Datenbank, 17
Blockhash-Matcher, 17
Blockhashers, 17
BMBF, 8

Chunk-Size, 9

DFXML, 7
digitale Spur, 7

EnCase, 7
Extensible-Storage-Engine-Format, 15

FTK, 7

NAS, 9

Open C3S, 8

PLY, 8

RAID, 9
RAID-Controller, 9
Rekall, 7

sqlite, 21
Stripe-Map, 9
Stripe-Size, 9
Striping, 9

Threema, 21
Thumb-Caches, 15
Thumbnails, 15
TSK, 7

Windows-Indexer, 15

X-Ways, 7