JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Parallelized File Carving

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:

Gregor Dorfbauer, BSc (0755807)

Angefertigt am:

Institut für Informationsverarbeitung und Mikroprozessortechnik

Beurteilung:

Assoz.Prof. Priv.-Doz. Mag. Dipl.-Ing. Dr. Michael Sonntag

Schwertberg, Februar 2013

Kurzfassung

Parallelized File Carving hilft, das Problem der Datenrekonstruktion von Festplattenabbildern zu beschleunigen. Die Größe von Festplatten steigt kontinuierlich, jedoch nimmt die Zugriffsgeschwindigkeit nicht in dem selben Maße zu.
Diese Masterarbeit hat zum Ziel, eine Architektur zur parallelen Analyse von forensischen Abbildern zu definieren und zu implementieren. Dadurch wird die Analysezeit eines Abbilds stark verkürzt. Durch ein Plugin-System können neue Dateitypen einfach hinzugefügt werden.
Die Arbeit gliedert sich in eine technische Motivation zum Thema File-Carving, der konkreten Umsetzung in Parallelized File Carving, sowie einer Beschreibung für den Benutzer und den Entwickler. Abschließend werden mögliche Erweiterungen dargestellt und der aktuelle Stand der Technik präsentiert.

Abstract

Parallelized File Carving accelerates data reconstruction from hard disk images. Hard disk drives' capacity increases steadily, however the access speed does not accelerate in the same way.
This master thesis targets at a new architecture for parallel analysis of forensic images. This architecture will be defined and implemented. The time needed for analyzing an image will be decreased. Using a plugin system, it's easy to add support for new file types.
This thesis includes a technical motivation regarding file carving and the use of file carving in this thesis. Additionally, there are manuals for users and developers. Finally, future improvements and related work are presented.

Danksagungen

An dieser Stelle möchte ich allen Menschen, welche mich während meines Studiums und speziell bei dieser Masterarbeit unterstützt haben, danken.

Ich bedanke mich bei meinem Betreuer Assoz.Prof. Priv.-Doz. Mag. Dipl.-Ing. Dr. Michael Sonntag für die tatkräftige Unterstützung und die wertvollen Anmerkungen und Ideen während der Bearbeitung dieser Masterarbeit.

Großer Dank gebührt meinen Eltern. Sie haben mich nicht nur während meines Studiums unterstützt, sondern gaben mir auch die Freiheit, diesen Teil meines Lebens selbst nach meinen Wünschen zu gestalten. Ich habe Hochachtung vor eurer Art, uns Kinder zu fördern, da ich in den letzten Jahren oft genug gesehen habe, dass diese Art der Unterstützung nicht selbstverständlich ist. Ich hoffe, dass ich in meinem späteren Leben ebenfalls diese unvergleichliche Fairness und Unterstützung weitergeben kann.

Ich bedanke mich bei Michael Mayrhofer für das Korrekturlesen dieser Arbeit sowohl auf orthographischer als auch technischer Ebene.

Danke, liebe Michaela für die Geduld während des Schreibens der Masterarbeit und dafür, dass Du mir immer geholfen hast, die Motivation aufrecht zu erhalten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

File Carving bezeichnet die Technik, aus einem Datenträgerabbild Dateien ohne die Hilfe der Struktur des verwendeten Dateisystems zu retten. Dabei können bereits gelöschte oder absichtlich versteckte Dateien für forensische Zwecke wieder gefunden werden. Dieser Prozess ist bedingt durch die größer werdenden Speichermedien sehr zeitintensiv.

Ziel dieser Masterarbeit ist es, den Vorgang des File Carving zu parallelisieren. Dabei sollen mehrere Prozesse auf einem Host, als auch mehrere Hosts gemeinsam, gleichzeitig Abbilder analysieren können. Das Blockdiagramm in Abbildung 1.1 illustriert den Aufbau. Der Vorgang soll über ein Web-Interface steuerbar sein. Über ein Plugin-System können verschiedene File-Carving-Algorithmen und -filter verwendet werden.

1.1 Aufgabenstellung

Um ein Abbild einer Festplatte zu analysieren, wird zuerst das Abbild forensisch ausgelesen und auf einem gemeinsamen Speicher (SAN[1], NAS[2]) abgelegt.

Im ersten Schritt (siehe Abbildung 1.2(a)) wird bei einem Carving-Job ein Abbild und allgemeine Einstellungen ausgewählt. Dabei kann die Task-Größe (wieviele Cluster werden auf einmal betrachtet), die Parallelisierung und die Verteilung auf den Clusternodes eingestellt werden. Der früheste Start ermöglicht die Planung von Tasks in der Zukunft. Im Web-Interface definiert man anschließend (Abbildung 1.2(b)), nach welchen Dateitypen (Filter bereitgestellt durch Plugins) gesucht werden soll. Jedes Plugin kann Parameter definieren, welche den Carving-Prozess steuern. Die Reihenfolge der Plugins ist frei wählbar.

Die erstellten Carving-Jobs (Abbildung 1.3(a)) können im Betrieb verwaltet werden. Bereits laufende Jobs sollen pausiert oder abgebrochen werden können.

[1] Storage Area Network
[2] Network Attached Storage

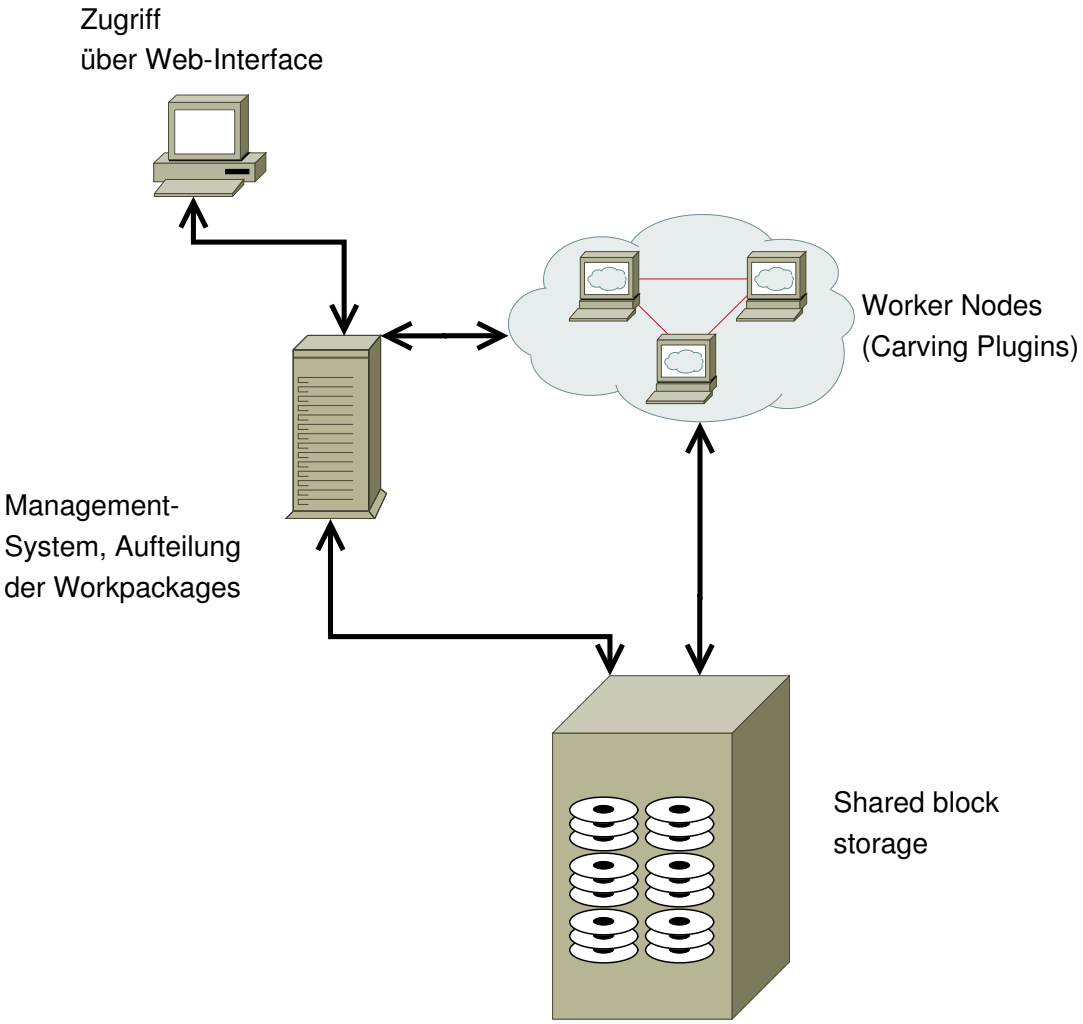
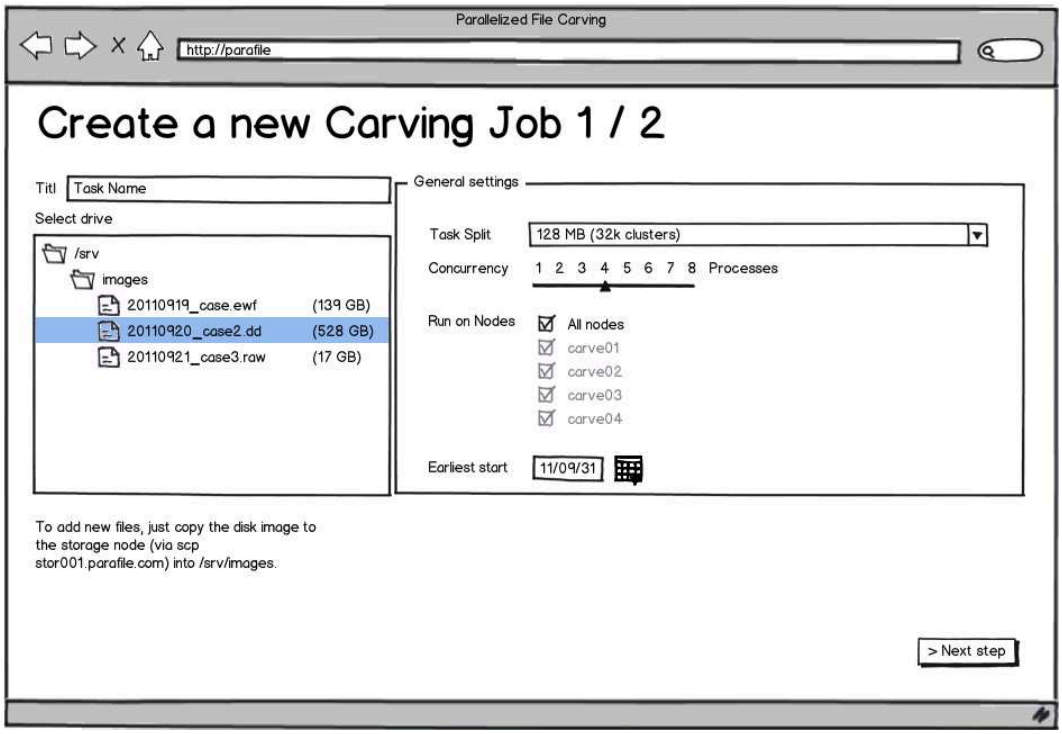


Abbildung 1.1: Blockdiagramm Parallelized File Carving

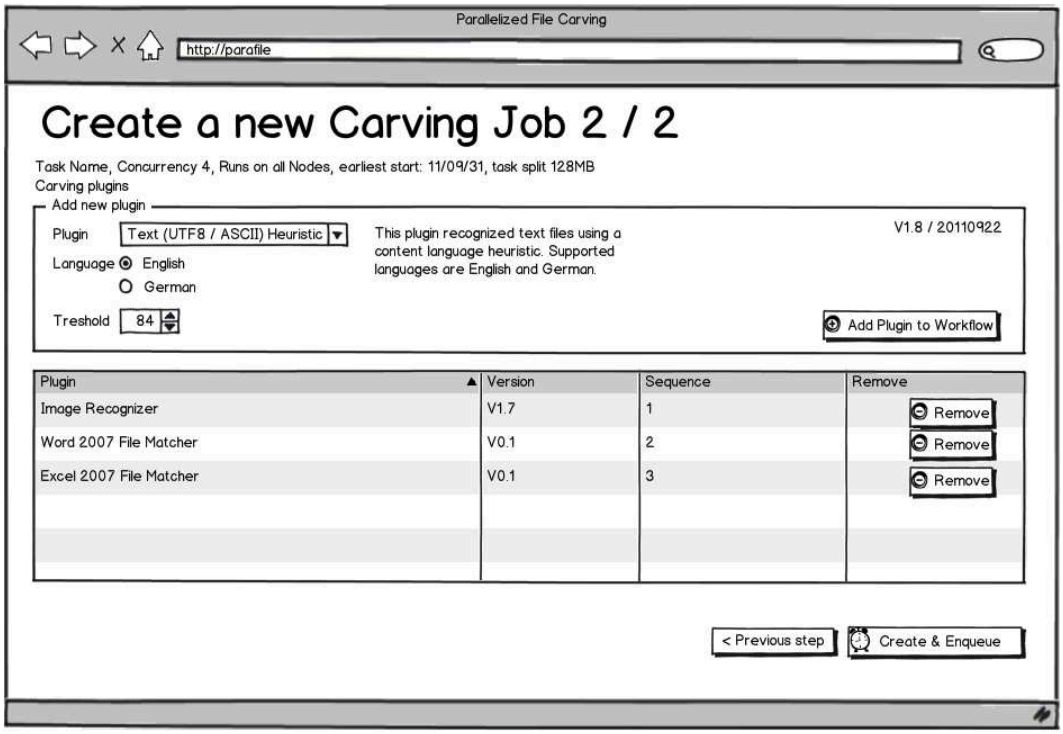
Eine Benachrichtigung per E-Mail, sobald ein lang andauernder Durchlauf abgeschlossen ist, soll einstellbar sein. Es können auch alle Jobs auf einmal gestartet (bzw. eingereiht), pausiert oder gestoppt werden.

Bestehende und abgeschlossene Jobs können auch als Vorlagen verwendet werden. Damit kann ein Abbild noch einmal mit anderen Plugin-Einstellungen untersucht werden, oder bestimmte Plugin-Einstellungen auf ein anderes Abbild angewandt werden.

Sobald Ergebnisse vorliegen, können gefundene Dateien in einer Übersicht angesehen werden. Neben den Metadaten können die Dateien hier auch zum Betrachten heruntergeladen werden (Abbildung 1.3(b)). In der Detailansicht (Abbildung 1.4) können

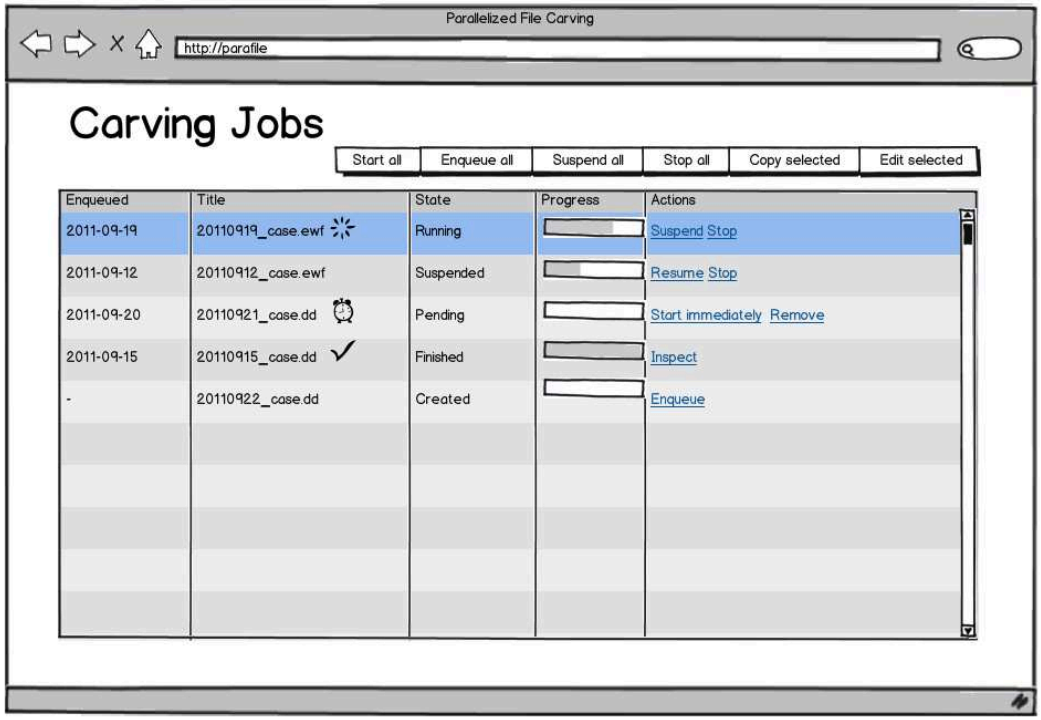


(a) Erstellen eines neuen Carving-Jobs, Schritt 1

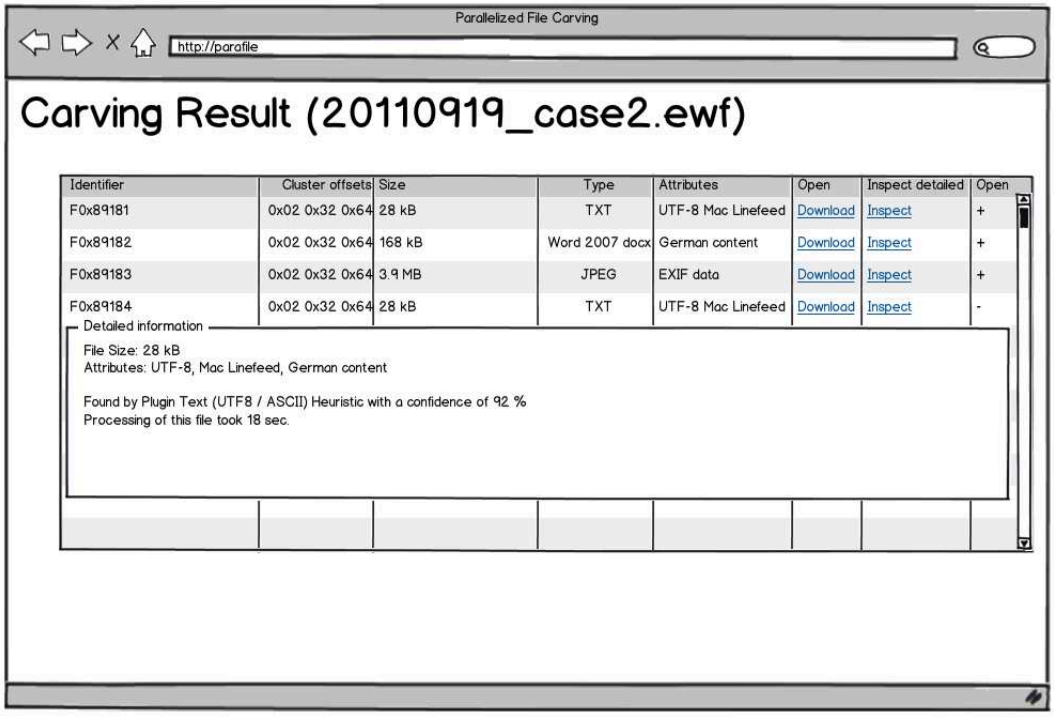


(b) Erstellen eines neuen Carving-Jobs, Schritt 2

Abbildung 1.2: Erstellen eines neuen Carving-Jobs



(a) Übersicht der Carving-Jobs



(b) Ergebnisse eines Carving-Jobs

Abbildung 1.3: Übersicht und Ergebnisse von Carving-Jobs

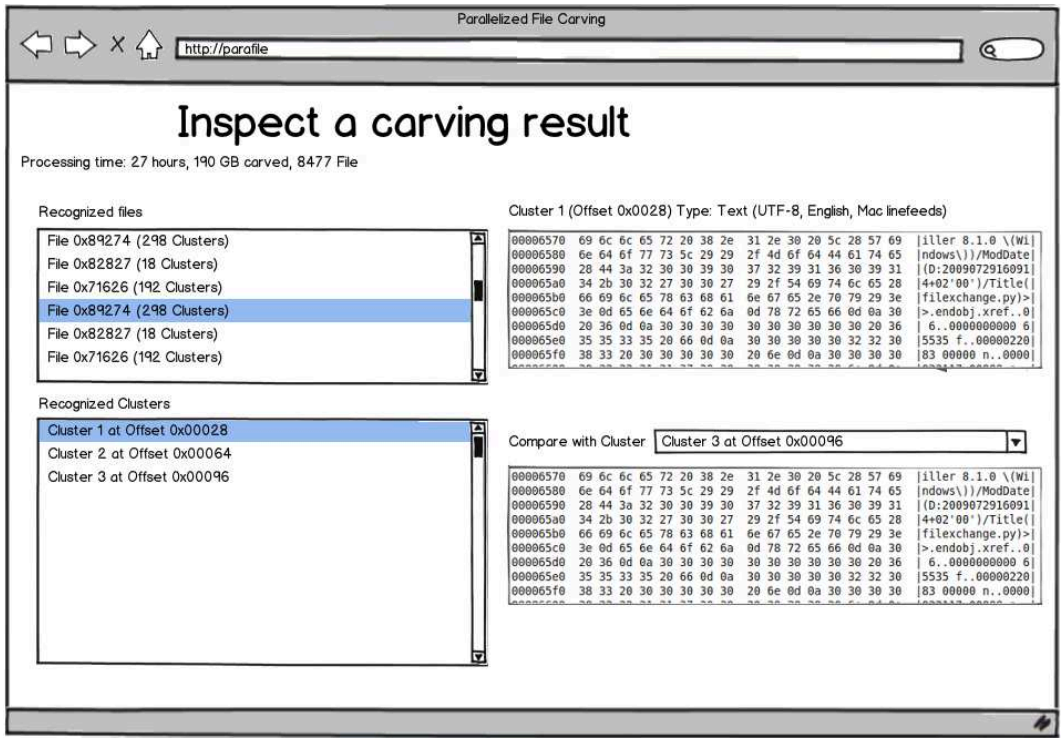


Abbildung 1.4: Detailansicht einer Datei

einzelne Cluster miteinander verglichen werden und manuell entschieden werden, ob die Cluster korrekt erkannt wurden oder zu anderen Dateien gehören.

Die Liste der erkannten Dateien bestimmt dabei die Liste der Cluster in dieser Datei. In der oberen Ansicht wird der ausgewählte Cluster angezeigt, in der unteren Ansicht kann ein beliebiger Cluster aus der Datei (oder dem Image) verglichen werden.

1.1.1 Ansatz der Parallelisierung

Die bereitgestellten Festplattenabbilder werden in gleich große Tasks von z.B. 128MB (entpricht 32768 Clustern bei einer Clustergröße von 4kB) aufgeteilt (siehe Abbildung 1.5). Prinzipiell kann jeder Carving-Node jedes Plugin auf jedes Taskset anwenden.

Die Reihenfolge der einzelnen Tasks und Plugins wird durch einen Scheduling-Algorithmus bestimmt. Bestimmte Plugins können auf Metadaten vorheriger Plugins aufbauen (z.B. wenn der Dateityp eines Clusters schon zweifelsfrei feststeht, muss er nicht mehr auf JPEG untersucht werden). Plugins erzeugen für jeden Cluster logische Informationen (z.B. Entropie, erkannte Sprache, Dateityp). Nachdem ein Plugin einen Task abge-

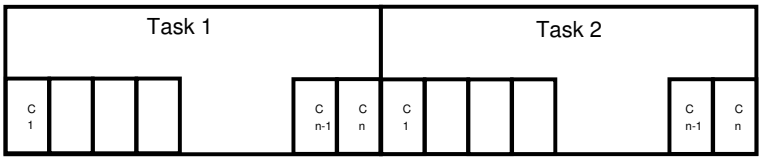


Abbildung 1.5: Aufteilung eines Tasks in verschiedene Cluster

schlossen hat, existieren logische Ergebnisse für jeden Cluster. Diese werden jetzt mit den benachbarten Clustern verglichen. Hat z.B. der vorherige Cluster kein Dateiende gefunden, bzw. passt die Entropie des letzten Clusters (Cn) zu der des ersten Clusters des Folgetasks (C1), können die Cluster zu einer Zieldatei verbunden werden.
Alle Tasks können also parallel ohne spezielle Kommunikation abgearbeitet werden.

Die Abarbeitung eines Tasks ist „atomar“. Sie können nur in einem Prozess und auf einem Rechner abgearbeitet werden. Wenn ein Job gestoppt oder pausiert werden soll, muss also nur auf die gerade bearbeiteten Tasks gewartet werden.

1.2 Aufbau dieser Arbeit

Diese Masterarbeit ist in folgende Teile strukturiert:

- Kapitel 2 beschreibt die notwendigen Grundlagen, um die Vorgehensweise beim File Carving zu verstehen. Es wird auf die Definition und den forensischen Ablauf von dem Beweisstück Festplatte bis zur Abbilddatei eingegangen. Die Techniken des File Carvings werden mit einem Beispiel zum Carven einer PNG-Datei erklärt.
- Im Kapitel 3 werden fortgeschrittene Ansätze zum File Carving präsentiert. Dabei werden Statistiken zum Finden von Dateien ohne weitere Strukturinformation und Möglichkeiten zum Reduzieren der zu untersuchenden Ergebnisse dargestellt.
- Der Stand der Technik wird im Kapitel 4 zusammengefasst, dabei wird speziell auf ähnliche Ansätze, d.h. verteiltes File-Carving und Multithreading auf der Graphikkarte eingegangen.
- Das Kapitel 5 erklärt den praktischen Teil dieser Masterarbeit. Naben den unterstützen Abbildformaten wird hier die Verteilung auf mehrere Systeme im Detail erklärt. Das Beispielplugin für Parallelized File Carving, welches Bilddateien im PNG- und JPEG-Format findet, wird hier vorgestellt. Die automatische Vorschau mit CarvFS und Geschwindigkeitsvergleiche schließen diesen Abschnitt ab.
- Im Benutzerhandbuch (Kapitel 6) wird die Installation von Parallelized File Carving auf einem Linux-System und die Bedienung des gesamten Programms gezeigt.
- Das Entwicklerhandbuch im Kapitel 7 zeigt, wie zusätzliche Plugins entwickelt, getestet und beschleunigt werden können.
- Im Kapitel 8 erfolgt eine Zusammenfassung der Inhalte dieser Masterarbeit.
- Mögliche Erweiterungen werden im Kapitel 9 skizziert. Neben Ideen für File Carving-Plugins gibt es auch Vorschläge für die Erstellung einer Editor-Oberfläche auf Blockebene und die Integration von GlusterFS, um die Ein- und Ausgabegeschwindigkeit zu erhöhen.

2 File Carving

2.1 Definition

Michael I. Cohen definiert in (Cohen, 2007) File Carving wie folgt:

> „File Carving - auch unter Data Carving bekannt - ist ein wichtiges Werkzeug für die digitale Forensik. Dabei wird versucht, Dateien aus einem Festplattenabbild ohne Kenntnis der Dateisysteminformationen zu rekonstruieren.“

2.2 Forensischer Ablauf

Bevor Dateien in einem Abbild gesucht werden können, muss die zu untersuchende Festplatte forensisch gesichert werden. In (Sonntag, 2009) wird dabei folgender Ablauf empfohlen:

Jeder lesender Zugriff auf die Daten über das normale Dateisystem verursacht im Allgemeinen Schreibzugriffe, da Zugriffszeitstempel auf die Festplatte geschrieben werden. Deshalb müssen Schreibzugriffe vermieden werden (durch Hardware- oder Software-Schreibblocker). Während oder nach dem Auslesen des Abbilds muss auch eine kryptographische Prüfsumme (Hash, z.B. SHA512) erstellt werden, damit später die Integrität des Abbilds nicht angezweifelt werden kann.

Das erstellte Abbild, liegt je nach verwendetem Programm als 1:1-Abbild (Raw-Image, erstellt durch Tools wie *dd*), als EWF-Abbild (Expert Witness Compression Format, Dateiformat erstellt durch *EnCase Forensics* der Firma Guidance Software oder der Open-Source-Lösung *libewf*) oder als AFF-Abbild (Advanced Forensics Format, entwickelt von Garfinkel et. al. (Garfinkel, 2006)) vor.

2.3 Abbildformate

2.3.1 Raw-Abbild

Raw-Abbilder, auch 1:1-Abbilder genannt, sind am einfachsten zu erzeugen. Die Festplatte wird direkt Byte für Byte in ein Abbild kopiert, ohne dabei zu komprimieren oder eine Prüfsumme zu berechnen.

Das Programm `dd` aus den GNU coreutils (GNU, 2012) sei exemplarisch als Erzeuger von Raw-Abbildern genannt.

Solche Abbilder können von nahezu jedem Carving-Werkzeug weiter verwendet werden. Die Nachteile dieser Raw-Abbilder sind der Platzbedarf (auch leere Stellen der Festplatte werden kopiert und es findet keine Kompression statt) und der fehlende Integritätsschutz durch Prüfsummen.

2.3.2 EWF-Abbild

Das Expert Witness Compression Format ist ein proprietäres Format der Firma Guidance Software und wird in dem Softwareprodukt EnCase Forensics eingesetzt.

Laut der Spezifikation (Metz, 2012) kann EWF sowohl für Festplatten als auch Partitionen eingesetzt werden. Dabei kann gewählt werden, ob die Daten komprimiert gespeichert werden sollen.

Neben EnCase Forensics wird EWF auch von FTK der Firma Access Data (AcessData, n.d.), und der Open-Source-Lösung libewf (Metz, n.d.) unterstützt.

Die Vorteil von EWF sind die breite Unterstützung in kommerziellen und freien Forensikwerkzeugen und die unterstützte Kompression und Integritätsprüfung.

EWF ist ein proprietäres Format, d.h. Formatänderungen seitens der Firma Guidance Software können zu Inkompatibilitäten zu anderen Systemen führen, wenn keine neuen Spezifikationen veröffentlicht werden.

2.3.3 AFF-Abbild

Das Advanced Forensics Format wurde von Garfinkel et. al. (Garfinkel, 2006) entwickelt, um ein offenes Format mit ähnlichem Funktionsumfang wie EWF zur Verfügung zu stellen.

Wichtige Designziele und Anforderungen an AFF waren (Garfinkel, 2006):

- Offenes Format: Proprietäre Formate wie EWF sind als Entwickler von Drittsoftware schwierig zu implementieren. Das Format ist komplex, und viele Details sind reverse engineered, d.h. ohne genaue Kenntnis der Spezifikation erstellt.
- Einfache Formate (Raw-Abbilder) erzeugen sehr große Abbilder, da sie unkomprimiert speichern. Weiters fehlt die Unterstützung von Metadaten, Signaturen und Prüfsummen.
- Traditionelle Abbildformate unterstützen oft nur einen Datenstrom. Oft sind mehrere Datenquellen (verschiedene Festplatten in einem System, USB-Sticks, ...) vorhanden, welche als eine logische Einheit betrachtet werden sollen.

Bestehende Raw-Abbilder können mit dem Programm `affconvert` in dieses Format umgewandelt werden. Dabei kann die integrierte Kompression und Hashwert-Erzeugung verwendet werden, um auch bestehende Abbilder platzsparend abzuspeichern. Das AFF-Abbild enthält dann sowohl die komprimierten Rohdaten als auch Metadaten zu diesem Abbild.

2.4 Sicherung der Integrität und Authentizität der Abbilder

Nach dem Auslesen des Abbilds wird die weitere Untersuchung anhand der Abbilddateien vorgenommen. Damit die gefundenen Schlüsse beweiskräftig sind, muss sichergestellt werden, dass das Abbild exakt mit dem Original übereinstimmt. Ansonsten könnte angenommen werden, dass bei der Ermittlung Daten verändert wurden.

Dazu kommen kryptographische Hashfunktionen und Signaturen zum Einsatz. Der folgende Abschnitt über Kryptographie wird in (Scharinger, 2012) vertiefend erklärt.

2.4.1 Kryptographische Hashfunktionen

Eine kryptographische Hash-Funktion ist eine Einwegfunktion H, welche aus einer beliebigen Menge von Daten M (*pre-image*) eine Ausgabe von fixer Länge h (*residue*) erzeugt.

$$h = H(M) \tag{2.1}$$

Dieser Fingerabdruck beschreibt den Inhalt des Abbilds kompakt, geringste Änderungen an dem Abbild sollen den Hashwert sofort verändern.

Damit eine Hashfunktion der Länge N als kryptographisch sicher gilt, müssen folgende drei Bedingungen gelten:

- Einwegeigenschaft (Preimage resistance)
  Der Hashwert h ist aus einem gegebenen Abbild M einfach zu berechnen. Bei einem gegebenen Hashwert h ist es jedoch schwer, ein Abbild M zu finden, welche $h = H(M)$ erfüllt.

  Die Hashfunktion ist sicher, wenn der Aufwand dazu bei $O(2^N)$ liegt.

- Schwache Kollisionsresistenz (Second preimage resistance)
  Bei einem gegebenen Abbild M ist es schwer, ein Abbild M' zu finden, so dass gilt $H(M) = H(M')$.

  Die Hashfunktion ist sicher, wenn der Aufwand dazu bei $O(2^N)$ liegt.

- Starke Kollisionsresistenz
  Es ist schwer, beliebige Abbilder M, M' zu finden, so dass gilt $H(M) = H(M')$.

  Die Hashfunktion ist sicher, wenn der Aufwand dazu bei $O(2^{\frac{N}{2}})$ liegt.

Hashfunktionen, welche als kryptographisch sicher gelten, sind die Funktionen der SHA-2-Familie (SHA-256, SHA-384 und SHA-512). Die Hashfunktionen MD5 und SHA1 sind häufig in Gebrauch, gelten jedoch als kryptographisch unsicher, da der Aufwand, um Kollisionen zu finden, bereits (MD5: stark) reduziert wurde.

2.4.2 Digitale Signaturen

Der errechnete Hashwert (siehe letzter Abschnitt) sichert die Integrität des Abbilds, das bedeutet, dass jede Änderung des Abbilds sofort einen anderen Hashwert bedingt. Der Hashwert an sich ist jedoch noch nicht kryptographisch gesichert und die Authentizität somit nicht gewährleistet.

Digitale Signaturen ermöglichen es, durch kryptographische Algorithmen sicher zu stellen, dass der Hashwert zu einem bestimmten Zeitpunkt und von einer bestimmten Person erstellt wurde und somit authentisch ist.

Bei Digitalen Signaturen wird der errechnete Hashwert mit dem privaten Schlüssel eines Public-Private-Schlüsselpaars verschlüsselt. Die Authentizität kann durch das Entschlüsseln mit dem öffentlichen Schlüssel (welcher durch ein Zertifikat einer anerkannten Zertifizierungsstelle authentifiziert wird) geprüft werden.

Bei der Prüfung der digitalen Signatur wird der Hashwert aus dem Inhalt neu generiert und mit dem entschlüsselten Hashwert der digitalen Signatur verglichen. Stimmen beide Werte überein, ist das Abbild unverändert.

2.5 Techniken des File Carvings

Nachdem das Abbild forensisch ausgelesen und die Integrität und Authentizität gesichert wurde, beginnt die eigentliche Aufgabe des File Carvings.

Cohen definiert in (Cohen, 2007) das Grundproblem des File-Carvings wie folgt:

Es wird versucht, den zu einer Datei gehörenden Byte-Strom aus dem Festplattenabbild zu extrahieren. Dabei wird eine Mapping-Funktion (siehe Abbildung 2.1) gesucht, welche Bytes im Abbild zurück auf Bytes in der Datei zusammensetzt.

2.5.1 Mapping-Funktion

Die Mapping-Funktion (siehe Abbildung 2.1) beschreibt die Position der Dateifragmente und die Fragmentgrenzen innerhalb eines Abbilds. Diese Funktion weist folgende Eigenschaften auf:

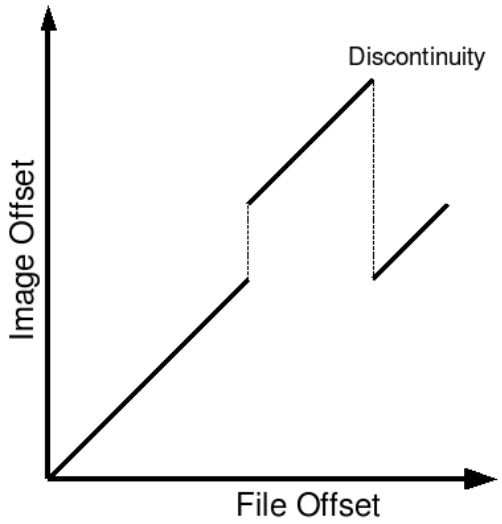


Abbildung 2.1: Beispiel einer Mapping-Funktion (Cohen, 2007)

- Die Steigung der Funktion ist immer 1, da eine 1:1 Abbildung des Abbilds und der Datei existiert.
- Es gibt Unstetigkeiten (Discontinuities) der Funktion (durch Dateifragmentierung hervorgerufen).
- Die Funktion ist invertierbar, jede Stelle ist maximal einer Datei zugeordnet. Das bedeutet, dass Teile, welche bereits zugeordnet sind, später nicht mehr betrachtet werden müssen.
- Im Allgemeinen sind die Unstetigkeiten an Sektorgrenzen (512 Byte) ausgerichtet.

Für ein endliches Festplattenabbild ist die Anzahl der verschiedenen Mapping-Funktionen durch die Permutation aller Sektoren limitiert. Die Ordnung der Permutation liegt bei $\mathcal{O}(n!)$. Ein Ausprobieren der Möglichkeiten ist damit bei aktuellen Kapazitäten von mehreren Terabyte nicht praktikabel.

Nachdem eine Mapping-Funktion (also eine Permutation des Festplattenabbilds, welche die Fragmente der Dateien in der richtigen Reihenfolge abbildet) gefunden wurde, benötigt man noch eine Diskriminator-Funktion (Cohen, 2007).

2.5.2 Diskriminator-Funktion

Die Diskriminator-Funktion überprüft die Gültigkeit einer gegebenen Mapping-Funktion. Der Diskriminator kann dabei mit verschiedenen Strategien vorgehen.

Eine Möglichkeit ist, das Format des Ergebnisses zu überprüfen. Stimmt der Aufbau der gefundenen Datei mit einer JPEG-Datei überein, ist die Wahrscheinlichkeit gegeben, dass die Datei tatsächlich ein JPEG-Bild ist.

Alternativ kann der gefundene Inhalt direkt analysiert werden. Besteht der Korpus der Datei hauptsächlich aus druckbaren ASCII-Zeichen, kann davon ausgegangen werden, dass die gefundene Datei ein Textdokument ist.

Abschließend gibt es die Möglichkeit einer externen Überprüfung, indem die Datei mit einem zusätzlichen Programm geöffnet und dekodiert wird. Beispiel hierfür ist ein mutmaßliches PDF-Dokument, welches erfolgreich von einem PDF-Anzeigeprogramm geöffnet werden konnte.

2.5.3 Aufbau eines File-Carvers

Ein File-Carver, bestehend aus einem Generator für Mapping-Funktionen und einem Diskriminator, ist wie in Abbildung 2.2 dargestellt, aufgebaut.

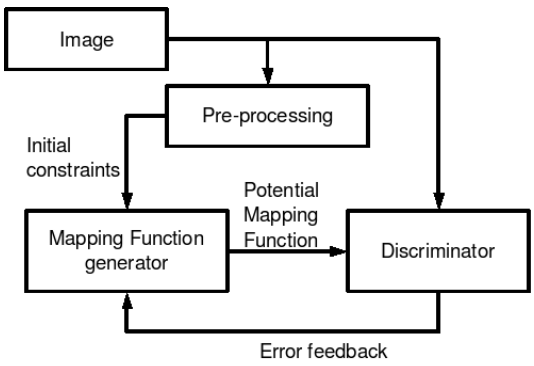


Abbildung 2.2: Aufbau eines File-Carvers (Cohen, 2007)

Mittels einer Vorverarbeitung (Pre-Processing) werden grundlegende Koordinaten für die Mapping-Funktion gefunden (als Beispiel seien magic bytes am Dateianfang genannt, welche die Mapping-Funktion abgrenzen).

Der Generator für Mapping-Funktionen erstellt aus diesen Basispunkten mögliche Fortsetzungen des Bytestroms und gibt diese an den Diskriminator weiter.

Der Diskriminator überprüft die Gültigkeit der Mapping-Funktion, welche gegeben ist, wenn eine Datei aus dem gefundenen Abschnitt rekonstruiert werden kann.

Sollte der Diskriminator Gründe für das Scheitern der Dekodierung gefunden haben, werden diese Fehler an den Generator gemeldet, um das Ergebnis der abgeschätzten Mapping-Funktionen zu verbessern (Error feedback).

2.6 Fragmentierung

Memon zeigt in (Memon & Pal, 2006) ein Beispiel für Dateifragmentierung. Eine Datei, welche größer als die verwendete Cluster-Größe des Dateisystems ist, wird potentiell fragmentiert abgespeichert.

Abbildung 2.3 zeigt ein Beispiel einer Festplatte mit zehn Clustern. In Abbildung

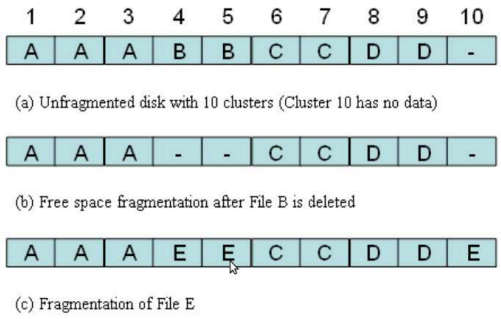


Abbildung 2.3: Fragmentierung von Dateien (Memon & Pal, 2006)

2.3 (a) sind vier Dateien A, B, C, und D fortlaufend und nicht fragmentiert abgespeichert. Im Schritt (b) wird die Datei B gelöscht und somit entsteht freier Speicherplatz zwischen A und C. Im nächsten Schritt (c) wird eine Datei E mit der Länge von drei Clustern abgespeichert, welche fragmentiert wird.

Das Dateisystem beinhaltet Informationen in einer Datenstruktur, welche dazu verwendet werden, um diese Fragmente wieder zusammenzufügen. Datei E ist mit Hilfe der Metainformationen also wieder vollständig rekonstruierbar. Sind diese Informatio-

nen jedoch nicht mehr zugänglich, da die Datei gelöscht wurde oder diese Datenstruktur nicht mehr existent ist, sind die Fragmente nicht mehr einfach zuordenbar.

Die Herausforderung beim File Carving liegt darin, ohne weitere Metainformationen des Dateisystems, die Dateien A, E, C und D zu rekonstruieren.

2.6.1 Fragmentierung in neuen Dateisystemen

Eine Analyse von gebrauchten Festplatten zeigt eine durschnittliche Fragmentierung von 6% der einzelnen Dateien (Garfinkel, 2007). Die Fragmentierung ist jedoch nicht gleichmäßig auf alle untersuchten Festplatten verteilt. So hatten die Hälfte der untersuchten rund 300 Festplatten keine einzige fragmentierte Datei.

Laut Garfinkel gibt es folgende Gründe für hohe Fragmentation:

- Dateien, welche häufig erweitert werden (Log-Dateien, Datenbanken, E-Mail-Dateien (Outlook PST)), werden zu groß für den ursprünglich reservierten Speicherplatz. Diese Dateien müssen in einem neuen Fragment fortgesetzt werden.
- Systemdateien, werden durch Updates verändert (gepatcht) und verändern sich dadurch in der Größe.
- Dateifragmentation ist bei Datenträgern, welche vollständig gefüllt waren, höher. Die zwischenzeitlich gelöschten Dateien stellen den freien Speicherplatz nur lückenhaft zur Verfügung.

2.7 Carven einer PNG-Datei

Das PNG (Portable Network Graphics) Format (Duce, 2003) ist ein weit verbreitetes, verlustlos komprimiertes Bitmapformat. Exemplarisch für andere Dateiformate wird das Carven eines PNG-Bildes aus einem Dateisystemabbild erklärt.

Folgende Informationen über den Aufbau des PNG-Formats sind dabei notwendig:

- Die Start-Sequenz (Magic Bytes) des PNG-Formats lautet hexadezimal `89 50 4E 47 0D 0A 1A 0A`. Die Sequenz `50 4E 47` bedeutet `PNG` im ASCII-Code.
- Das PNG-Format ist in Blöcke (sogenannte *Chunks*) unterteilt. Chunks bestehen aus einem 8 Byte Header (4 Byte Länge als Big-Endian Integer, 4 Byte Typ als

Character-Array), dem eigentlichen Datenblock und einer CRC-Prüfsumme (4 Byte).

- PNG-Dateien beginnen stets mit einem `IHDR`-Chunk und enden mit einem `IEND`-Chunk.

```
1  def _checkPNG(f, off):
2    inoff = off
3    maxoff = off + MAX_SIZE
4    f.seek(off)
5    data = f.read(8)
6    off += 8
7    if data == "\x89PNG\r\n\x1A\n":
8      block = ""
9
10     while block != "IEND" and off < maxoff:
11       data = f.read(8)
12       off += 8
13       if len(data) == 8:
14         length = struct.unpack(">I", data[:4])[0]
15         if length >= 0:
16           block = data[4:]
17           off += length + 4
18       f.seek(off)
19
20     if off >= maxoff:
21       return inoff
22     else:
23       return off
24 else:
25   return inoff
```

Listing 2.1: Beispiel eines PNG-Carvers

Das Listing 2.1 zeigt einen einfachen PNG-Carver, welcher in einer geöffneten Datei `f` nach PNG-Dateien sucht. Diese Carving-Funktion geht davon aus, dass die PNG-Dateien nicht fragmentiert abgespeichert sind.

In den Zeilen `2-6` werden zuerst die Suchgrenzen der Datei mit einer künstlich eingeführten Maximaldateigröße `MAX_SIZE` begrenzt und die ersten 8 Byte der Datei eingelesen. Wenn die Magic Bytes übereinstimmen (d.h. eine PNG-Datei gefunden ist),

wird das PNG-Format dekodiert. Dabei wird ab Zeile 10 der PNG-Header solange eingelesen, bis der letzte Block (`IEND`) gefunden wurde. Ist dies nicht der Fall, wird anhand der Blocklänge der nächste Block-Header gesucht (Zeilen `13-17`).

Wenn der letzte Block gefunden wurde und man sich noch innerhalb der gesetzten Grenzen befindet, wird das Ende der PNG-Datei als Offset in Zeile `23` zurückgegeben.

Wurde keine PNG-Daten gefunden, wird der Start-Index zurückgegeben und der Benutzer der Funktion kann bestimmen, dass es sich um keine PNG-Datei handelt.

3 Fortgeschrittene File Carving Techniken

Im Kapitel 2 wurden grundlegende Techniken des File Carvings dargestellt. Im einfachsten Fall gehen diese Ansätze von einer nicht fragmentierten Datei aus. Oftmals sind Dateien jedoch in verschiedenen Teilbereichen eines Abbilds abgespeichert. Solche Dateien können nur durch fortgeschrittene File Carving-Techniken gefunden werden.

Wie in Abschnitt 2.5 gezeigt, kann eine Mapping-Funktion unstetige Stellen aufweisen. Die Grenzen der Unstetigkeiten korrespondieren mit unterschiedlichen Fragmenten einer Datei. Ziel dieser fortgeschrittenen Techniken ist nun, die Grenzen der jeweiligen Inhalte durch eine Inhaltsanalyse auf Blockebene zu finden.

Abschnitt 3.1 präsentiert statistische Ansätze, um Inhaltsgrenzen zu bestimmen. Ein motivierendes Beispiel hierzu ist die wechselnde Entropie von einem komprimierten Archiv hin zu einem Teil mit Klartext. Wechselt die Entropie, ist auch ein Wechsel des Dateiinhalts wahrscheinlich.

Abschnitt 3.2 beschäftigt sich mit dem Wiederaufsetzen der Suche nach einer Datei. Sobald ein Ende erkannt wurde, soll durch diese Technik auch der restliche Teil der Datei im verbleibenden Abbild entdeckt werden.

Das Proof of Discriminator-Verfahren in Abschnitt 3.3 zeigt einen Ansatz, um Dateifragmente durch eine Dekodierung zu validieren. Lässt sich eine gefundene Menge von Fragmenten vollständig, dem Dateiformat folgend, dekodieren, steigt die Wahrscheinlichkeit, dass die Datei vollständig und korrekt erkannt wurde.

Abschließend wird im Abschnitt 3.4 eine Optimierung skizziert, welche bereits bekannte Dateien (Systemdateien, welche in vielen Abbildern ident vorkommen) aus der späteren Analyse ausschließt, um die Bearbeitungsdauer zu minimieren.

3.1 Statistische Ansätze

Cor J. Veenman zeigt in (Veenman, 2007) einen statistischen Ansatz, um Blöcke zu klassifizieren. Das verwendete statistische Lernen soll den Unterschied zwischen den Änderungen innerhalb einer Datei (Farbe innerhalb eines Bilds ändert sich) und den Änderungen zwischen zwei Dateien (auf eine Textdatei folgt ein komprimiertes ZIP-Archiv) erkennen.

Der erste Schritt ist die Definition von geeigneten Features, welche die Klassifizierung der Cluster ermöglicht. Diese charakteristischen Eigenschaften umfassen ein Byte-Histogramm, die Entropie und die Kolmogorov-Komplexität (Kolmogorov, 1965).

- Das Byte-Histogramm ist unabhängig von der Reihenfolge und zeigt die Häufigkeit der einzelnen Zeichenwerte. Aus diesem Fingerabdruck kann bestimmt werden, ob nur Klartext in einer Datei vorkommt.
- Die Entropie wird aus dem Byte-Histogramm abgeleitet und zeigt den Informationsgehalt innerhalb des Blocks.
- Die Kolmogorov-Komplexität berücksichtigt, im Gegensatz zum Histogramm und der Entropie, die Reihenfolge der Zeichen innerhalb eines Blocks.

Veenman verwendet diese Features sowohl in einer Zwei-Klassen-Erkennung als auch in einer Mehrklassen-Erkennung. Die Zwei-Klassen-Erkennung beantwortet die Fragestellung, ob ein Block zu einem bestimmten Dateityp gehört. Die Mehrklassen-Erkennung klassifiziert einen Block direkt in eine Dateitypklasse.

Die Ergebnisse der Untersuchung zeigen eine erfolgreiche Erkennung von HTML- und JPEG-Dateien im Vergleich zu anderen Binärdateien.

Calhoun und Coles verwenden in (Calhoun & Coles, 2008) ebenfalls statistische Ansätze und statistisches Lernen. Die Anzahl der Features und deren Komplexität ist höher als bei Veenman, jedoch ist die Auswertung nur als Zwei-Klassen-Erkennung ausgeführt. Die Erkennungsgenauigkeit von Calhoun liegt in der selben Größenordnung wie die Genauigkeit der Zwei-Klassen-Erkennung bei Veenman.

3.2 Wiederaufsetzen

Sobald die Typen von Einzelfragmente bestimmt wurden (hier können die statistischen Ansätze aus 3.1 zum Einsatz kommen), müssen diese Fragmente in die richtige Reihenfolge gebracht und den einzelnen Dateien zugeordnet werden. Karresand hat in (Karresand & Shahmehri, 2008) eine Möglichkeit veröffentlicht, wie Fragmente von JPEG-Dateien wieder zugeordnet werden können.

Das JPEG-Format bietet die Möglichkeit, die Codierung innerhalb der Datei neu zu beginnen. An diesen Stellen werden Reset-Marker (RST) eingefügt. Dadurch führen Übertragungsfehler oder veränderte Bereiche nicht zu einem komplett zerstörten Bild.

Diese Reset-Marker ermöglichen es, die Reihenfolge der Fragmente zu rekonstruieren, da an der Stelle der Grauwert fix definiert ist. Führt die Dekodierung der vorangegangenen Daten zu dem selben Grauwert, passen diese Fragmente zusammen und die Reihenfolge ist korrekt.

Der Ansatz benötigt einige Voraussetzungen:

- Das JPEG-Bild muss mit Reset-Markern (`0xFF0*`) versehen sein.
- Das Intervall dieser Reset-Marker (RST) muss bekannt sein.
- Die verwendeten Huffman-Tabellen zur JPEG-Komprimierung müssen bekannt sein. Diese Tabellen sind oft am Anfang der Datei abgespeichert.
- Die Sampling-Faktoren müssen bekannt sein.

Karresand hat eine Auswahl von 76 Digitalkameras auf diese Eigenschaften hin untersucht. 69 der ausgewählten Kameras haben die selben Huffman-Tabellen verwendet - die im Standard vorgeschlagegenen Tabellen. Insgesamt 12 Kameras aus diesen 69 weisen auch die notwendigen RST-Marker auf, um ein Wiederaufsetzen der Dekodierung zu ermöglichen.

Zusätzlich werden Regionen mit starken Änderungen im Bild als Anhaltspunkt zur Abschätzung der Bildgröße verwendet. Bei einer vertikalen Kante wiederholt sich eine Region mit starker Änderung (Hintergrundmuster auf Vordergrundmuster) mit der Bildbreite.

Die Ergebnisse zeigen eine Erkennungsrate von 97% bei einer Fragmentmenge bestehend aus einem Bild und eine Erkennungsrate von 70% bei einer gemischten Fragmentmenge aus sechs Bildern.

3.3 Dekodieren / Proof of Discriminator-Funktion

Der Diskriminator ist eine Funktion, welche im Carving-Prozess überprüft, ob die aktuell betrachteten Blöcke zu einem bestimmten Dateityp gehören (siehe Abschnitt 2.5). Solche Diskriminatoren können Programme sein, welche das Dateiformat unterstützen, also der Adobe Reader für PDF oder Microsoft Excel für XLS (Cohen, 2007). Diese Programme sind jedoch für den produktiven Einsatz als Diskriminator nicht geeignet, da Sie bei fehlerhaften Eingaben entweder den Fehler ignorieren oder ohne genauere Informationen abbrechen.

Die Position des Fehlers bei der Dekodierung ist jedoch zur Bestimmung der Fragmentgrenzen notwendig, um eine bessere Mapping-Funktion zu finden.
Cohen zeigt in (Cohen, 2007) einen selbst entwickelten PDF-Diskriminator, welcher einen Teil der PDF-Spezifikation implementiert und hilfreiche Fehler zurückgeben kann, um die Mapping-Funktion zu optimieren. Dieser Diskriminator ermöglicht es, eine PDF-Datei Sektor für Sektor zu analysieren und somit vorzeitig Fragmentgrenzen zu erkennen.

In (Cohen, 2008) verwendet Cohen eine universelle Bibliothek (*libjpeg*) um einen JPEG-Diskriminator zu betreiben. Jede Zeile des Bilds wird dabei einzeln durch die Bibliothek dekodiert. Wird das JPEG-Format nicht eingehalten, tritt ein Dekodierungsfehler auf und der betreffende Sektor gehört nicht zu diesem Bild. Sollte die Zeile wider erwarten dennoch korrekt dekodiert werden, entscheidet eine Kantenerkennung, ob der Sektor zu diesem Bild passt. Starke Änderungen der Bildinformation, welche über normale Kanten hinausgehen, weisen auf ein Fragment hin.

3.4 Filtern von existierenden Dateien

Die gefundenen Dateien aus einem Dateisystemabbild müssen schlussendlich von einem Menschen inspiziert werden. Wünschenswert ist, dass bereits bekannte Dateien unterschiedlich hervorgehoben werden.

Vrubel fasst diese Dateien in (Vrubel, 2011) zu zwei Kategorien zusammen. Bereits bekannte Dateien, welche nicht für die Ermittlung relevant sind, bilden die erste Kategorie. Hierzu zählen Betriebssystemdateien, welche auf annäherend jedem Festplattenabbild zu finden sind und nicht weiter analysiert werden müssen. Das National Institute of Standards and Technology (NIST) veröffentlicht unter (NIST & Technology, 2012) umfangreiche Hash-Listen von in Standardsoftware enthaltenen Dateien in der National Software Reference Library.

Die zweite Kategorie umfasst bekannte Dateien mit kriminellen Inhalten. Die hier zugrunde liegenden Hash-Datenbanken werden im Zuge von Ermittlungen erstellt.

Nach der Klassifikation der gefundenen Dateien können *known-good*-Dateien ausgeblendet und *known-bad*-Dateien speziell hervorgehoben werden. Die für die Analyse benötigte Zeit kann so gesenkt werden und Dateien, welche unter vielen irrelevanten Dateien gefunden werden, werden weniger leicht übersehen.

4 Stand der Technik

Waren noch vor wenigen Jahren Festplatten mit einigen 100 GB im privaten und geschäftlichen Einsatz vorherrschend, sind heute Einzelplatten mit 4 TB verfügbar. Die Kapazität der Festplatten steigt exponentiell mit einer Verdoppelung alle 12 Monate (Kryder, 2005). Die Integrationsdichte von Mikrochips steigt ebenfalls exponentiell bei einer Verdoppelung alle 18 Monate (Moore, 1965).

In Kontrast dazu steigt die Zugriffsgeschwindigkeit nur linear (Roussev & Richard III, 2004). Dieses Problem kann gelöst werden, indem mehrere Festplatten parallel als Verbund (RAID) und mehrere Rechner als Cluster arbeiten.

Obwohl die Rechenleistung exponentiell wächst, steigt die Geschwindigkeit eines einzelnen Rechenkerns nicht annähernd so stark wie die Gesamtrechenleistung. Die Rechenleistung wird dabei durch Parallelisierung auf Chip-Ebene (Multi-Core-CPUs und -GPUs) erreicht.

Entwickler von File-Carving-Programmen müssen diese beiden Umstände, also die langsamer wachsende Zugriffsgeschwindigkeit der Festplatten und die massive Parallelisierung von Rechenkernen, beachten.

Exemplarisch für den Stand der Technik werden in diesem Abschnitt zwei Forschungsprojekte, welche auf Parallelisierung des File-Carvings ausgerichtet sind, vorgestellt.

4.1 Verteiltes File-Carving im Cloud-Umfeld

In (Koo, 2012) skizziert Bon Min Koo eine Cloud-Architektur zum Einsatz bei der digitalen Forensik. Die zu analysierenden Datenmengen steigen ständig, jedoch ist die Auslastung der zu betreibenden Analysehardware nicht konstant. Bei großen Aufträgen würde man eine hohe Anzahl an Rechenknoten benötigen, um die Analyse zeitnah durchführen zu können. Jedoch gibt es auch Zeiten, in denen diese anzuschaffende

Hardware nicht ausgelastet sein wird.

Eine skalierbare (mit den Anforderungen wachsende und schrumpfende) Architektur ermöglicht eine rasche Bearbeitung bei gleichzeitig optimalem Ressourceneinsatz. Dieser Abschnitt definiert zuerst den Begriff Cloud-Computing generell, anschließend wird Infrastructure as a Service am Beispiel von Amazon Web Services (AWS, 2012) erklärt. Koos Architektur schließt diesen Abschnitt ab.

4.1.1 Cloud-Computing

Christian Baun et. al. definieren Cloud Computing in (Baun, 2010) folgendermaßen:

> „Unter Ausnutzung virtualisierter Rechen- und Speicherressourcen und moderner Web-Technologien stellt Cloud Computing skalierbare, Netzwerkzentrierte, abstrahierte IT-Infrastrukturen, Plattformen und Anwendungen als on-demand-Dienste zur Verfügung. Die Abrechnung dieser Dienste erfolgt nutzungsabhängig."

In anderen Worten bietet Cloud-Computing eine Infrastruktur an, welche nach Bedarf in kurzer Zeit erweitert werden kann. Der Anwender bezahlt nur für die tatsächlich benötigte Leistung.

Abbildung 4.1 zeigt die möglichen Ausprägungen des Cloud-Computings. Die Schichten in diesem Architekturmodell können aufeinander aufbauen, sind jedoch auch ohne den darunter liegenden Schichten einsatzfähig.

Die grundlegende Schicht, das sogenannte Infrastructure as a Service (IaaS) bietet Infrastrukturdienste (Netzwerk, Speicherplatz) und Rechenkapazität in Form von virtuellen und physikalischen Maschinen an.

Platform as a Service richtet sich an Entwickler, welche eine komplette Programmierumgebung vorfinden. Die PaaS-Schicht wird mit einer Applikation gestartet und kümmert sich darum, dass diese auf genügend IaaS-Instanzen läuft, damit die Applikation funktioniert.

Für Endbenutzer ist die Software as a Service-Schicht am interessantesten. Der Anwender mietet die Applikationen, welche er benötigt, und muss sich nicht um Wartung oder Betrieb der darunterliegenden Schichten kümmern.

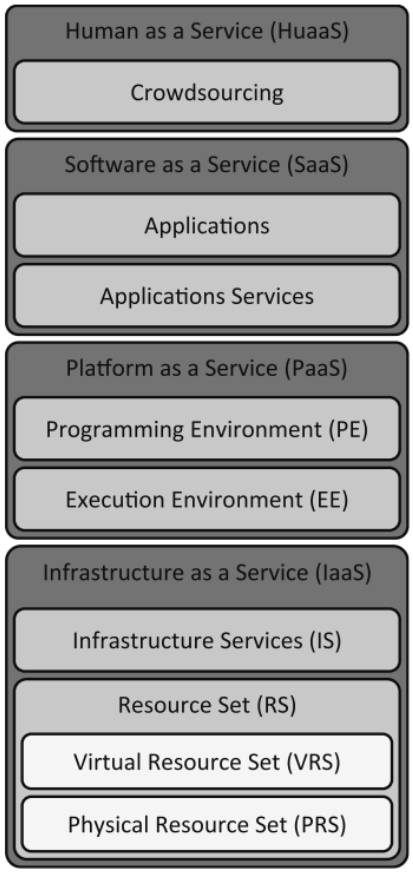


Abbildung 4.1: Schichtenmodell bei Cloud-Computing (Baun, 2010)

Die Human as a Service-Schicht schafft einen Markt für Aufgaben, welche aktuell nur schwierig maschinell automatisierbar sind. Dazu zählen unter anderem die Klassifikation von Bildern oder das Übersetzen von Texten. Diese Aufgaben werden dann von Personen durchgeführt.

4.1.2 Forensik in der Cloud

Koo stellt in (Koo, 2012) das Konzept des DFSaaS vor, dem Digital Forensic Software as a Service.
Die Ergebnisse und Dateien der forensische Beweissicherung direkt am Tatort und des Auslesens der Datenträger sollen direkt in Cloud-Speichern landen. Dadurch ist es möglich, in der Analysephase schnell auf diese Daten zugreifen zu können. Koo schlägt

als Entwicklungsframework MapReduce (Dean & Ghemawat, 2008) vor, um die Verarbeitung mit der Anzahl der verwendeten Cloud-Instanzen skalieren zu können.

Zusätzlich sollen auch virtualisierte Desktops in der Cloud zum Einsatz kommen, damit existierende Softwarelösungen weiterverwendet werden können. Hierbei ist jedoch zu beachten, dass die existierenden Werkzeuge nicht so stark von der Cloud-Infrastruktur profitieren können, da sie ursprünglich für Einzelplatzsysteme entwickelt wurden.

4.1.3 Abgrenzung zu Parallelized File Carving

Das vorgestellte Konzept DFSaaS liegt nur als Architekturskizze vor. Es werden konkrete Einsatzbeispiele genannt, welche zum Teil auf das Wiederverwenden von bestehenden Systemen in einem Cloud-Umfeld abzielen. Weiters ist der Upload der zu analysierenden Daten in den Cloud-Speicher zeitintensiv.

Es spricht nichts gegen die Verwendung von Parallelized File Carving im Cloud-Umfeld. Die zur Verfügung gestellte Infrastruktur ist zu der Architektur von Parallelized File Carving kompatibel und ermöglicht eine schnellere Bearbeitung durch das Hinzufügen von Rechenknoten im Cloud-Umfeld.

4.2 File Carving mit Graphikkarten

Lodovico Marziale zeigt in (Marziale, 2007) die Verwendung von hoch parallelisierten Prozessoren auf Graphikkarten, um File Carving zu betreiben. Der verwendete NVIDIA G80-Prozessor (GPU, graphical processing unit) wird durch die CUDA-Bibliothek für allgemeine Aufgaben, welche keine 3D-Graphiken erzeugen, zugänglich.

Die Aufgaben beim File Carving, also einen Datenstrom mit vielen verschiedenen Referenzmustern zu vergleichen, kann gut mit dieser Architektur umgesetzt werden. Marziale merkt an, dass die Suche parallelisierbar sein muss. Ein Abbild, welches gleichzeitig auf 30 verschiedene Dateitypen untersucht wird, profitiert mehr von dieser Parallelisierung, als ein Abbild, welches nur nach zwei verschiedenen Dateitypen untersucht wird.

Das zugrunde liegende Problem hierbei sind die Kopiervorgänge von der Host-CPU Richtung Graphikkarte und retour. Nur wenn ausreichend viele Operationen auf einem geladenen Datenblock ausgeführt werden können, rentiert sich der Einsatz der Graphikkarte zum File Carving.

5 Parallelized File Carving

Dieser Abschnitt beschreibt den praktischen Teil dieser Masterarbeit. Zuerst wird die konkrete Implementierung und die gewählte Softwarearchitektur gezeigt. Danach wird auf die unterstützten Dateiformate eingegangen. Dabei wird konkret erklärt, wie die Abbilder erzeugt werden.

Die Verteilung auf mehrere Hosts zeigt die Parallelisierung und die Synchronisation der beteiligten Rechenknoten. Die Aufteilung der Analyseaufgaben und das zur Verfügung stellen der Abbilddateien wird erklärt.

Das Beispiel-Plugin, welches Bilder im PNG- und JPEG-Format erkennt, wird detailliert beschrieben und zeigt somit, wie die einzelnen File Carving-Aufgaben umgesetzt werden.

Die gefundenen Dateien werden ohne zusätzlichen Speicherplatz zu verbrauchen extrahiert. Dabei kommt die Bibliothek CarvFS zum Einsatz, welche auch eine direkte Vorschau der Ergebnisse ermöglicht.

Abschließend wird der Speed-Up der Parallelisierung analysiert.

5.1 Implementierung von Parallelized File Carving

Parallelized File Carving besteht aus zwei Softwaresystemen. Die Steuerung und Erzeugung von Aufgaben wird in einem Webbrowser vorgenommen. Die Ausführung der Plugins findet im zweiten Softwaresystem im Hintergrund statt.

5.1.1 Webinterface

Die Oberfläche des Webinterface ist im JavaScript-Framework ExtJS implementiert (ExtJS, 2012). Die Oberfläche kommuniziert mit der Steuerungssoftware über eine JSON-Schnittstelle (Crockford, 2006). Sobald ein Carving-Job erzeugt wurde, wird die

Beschreibung für diese Aufgabe (Dateisystemabbild, gewählte Plugins und Einstellungen) an das zweite Softwaresystem übergeben.

Das Webinterface stellt auch die Konfiguration der Plugins zur Verfügung. Jeder Plugin-Parameter wird automatisch mit einer graphischen Oberfläche visualisiert, damit der Benutzer die Werte einfach setzen kann.

Model-View-Controller-Konzept

Die Ansichten werden durch das Model-View-Controller-Konzept von ExtJS bereitgestellt. Die Model-Komponente stellt das Datenmodell in sogenannten Stores, welche mittels JSON-Abfragen Daten erhalten, zur Verfügung. Views und Controller können in ExtJS zu einer View verschmolzen werden und greifen auf die jeweiligen Stores als Datenmodell zu.

Ein ExtJS-Store beinhaltet die definierten Felder mit den Datentypen und die dazugehörige JSON-URL, unter welcher die Daten abgerufen werden können. Dieses Konzept unterstützt Sortieren, Filtern und Paging auf der Serverseite. Zusätzlich könnten diese Funktionen auch direkt im Webinterface verarbeitet werden, wenn sichergestellt werden kann, dass die gesamten Daten bereits vorhanden sind.

Listing 5.1 zeigt exemplarisch die Definition eines ExtJS-Stores `PFC.store.FSStore` für den Dateisystembaum. Der Store ist abgeleitet von einem ExtJS-TreeStore und stellt das Datenmodell für eine Baumkomponente innerhalb einer View zur Verfügung. Die Datenfelder werden direkt mit Typ definiert, fehlt diese Angabe, wird automatisch ein String-Datentyp angelegt. Die Daten werden unter der relativen Adresse `/jobs/ls` abgerufen.

```
1  Ext.define('PFC.store.FSStore', {
2    extend : 'Ext.data.TreeStore',
3
4    constructor : function(cfg) {
5    var me = this;
6    cfg = cfg || {};
7    me.callParent([Ext.apply({
8      storeId : 'FSStore',
9      proxy : {
10     type : 'ajax',
11     url : '/jobs/ls',
```

```
12      reader : {
13       type : 'json',
14       root : 'ls'
15      }
16      },
17      fields : [{ name : 'file', type : 'string' },
18                 { name : 'size', type : 'int' },
19                 { name : 'changed', type : 'date' }
20                ]
21    }, cfg)]);
22   }
23  });
```

Listing 5.1: ExtJS-Store für Dateisysteme

Die zu diesem Store gehörende View-Komponente stellt den Dateisystembaum dar und bietet dem Benutzer einen Aktualisieren-Knopf. Durch die Definition des Stores und der Darstellung der jeweiligen Datenspalten werden automatisch die Daten angezeigt.

Das Listing 5.2 zeigt exemplarisch die Erstellung einer ExtJS-View `PFC.view.ui.FSTreePanel`, welche von der ExtJS-Klasse `Ext.tree.Panel` abgeleitet ist. Neben der Definition des zu verwendenden Stores `FSStore` werden auch die einzelnen Datenspalten mit den Überschriften und Breiten konfiguriert. In einer zu dieser Komponente gehörigen Toolbar am unteren Rand wird der Aktualisierungs-Button definiert.

```
1  Ext.define('PFC.view.ui.FSTreePanel', {
2     extend : 'Ext.tree.Panel',
3     height : 250, autoScroll : true,
4     store : 'FSStore',
5     initComponent : function() {
6      var me = this;
7
8      Ext.applyIf(me, {
9         viewConfig : { rootVisible : false },
10        columns : [
11          { xtype : 'treecolumn', dataIndex : 'file',
12           flex : 1, text : 'File' },
13          { xtype : 'gridcolumn', width : 125,
14           dataIndex : 'size', text : 'Size' },
15          { xtype : 'datecolumn', width : 175,
16           dataIndex : 'changed', text : 'Changed',
```

```
17              format : 'd.m.Y H:i:s' }
18          ],
19          dockedItems : [{
20              xtype : 'toolbar', itemId : 'toolbar',
21              dock : 'bottom',
22              items : [{ xtype : 'button', itemId : 'refreshButton',
23                         text : 'Refresh' }]
24            }]
25        });
26      me.callParent(arguments);
27    }
28  });
```

Listing 5.2: ExtJS-View für Dateisysteme

Zusätzlich zu der dieser reinen View-Definition gibt es eine Klasse, welche dieser View Funktionalität verleiht - in klassischen MVC-Architekturen würde diese Klasse als Controller bezeichnen werden. Das Listing 5.3 zeigt, wie der Reload-Button den Store neu vom Server laden lässt. Die Controller-Klasse leitet von der reinen Definitionsklasse ab und dient als Platz für Event-Handling und eigene Methoden.

```
1  Ext.define('PFC.view.FSTreePanel', {
2      extend : 'PFC.view.ui.FSTreePanel',
3      alias : 'widget.pfctreepanel',
4
5      initComponent : function() {
6        var me = this;
7        me.callParent(arguments);
8        this.toolbar = this.getComponent("toolbar");
9        this.refreshButton = this.toolbar.getComponent("refreshButton");
10       this.refreshButton.on("click", function() {
11           var store = Ext.StoreMgr.get("FSStore");
12           store.load();
13         }, this);
14     }
15   });
```

Listing 5.3: ExtJS-Controller für Dateisysteme

5.1.2 Backend

Das Backend der Applikation ist in Python implementiert und besteht aus der Schnittstelle zum Frontend (Webinterface) und der Verwaltung des Queuing-Systems. Dabei kommt ein Python Web-Framework zum Einsatz (Pylons, 2010).

JSON-Datenschnittstelle zum Frontend

Um das Beispiel des Frontends weiter zu führen, wird der Datenfluss des Dateisystembaums analysiert. Die oben konfigurierte URL `/jobs/ls` wird durch das Routing-System von Pylons auf den Controller `jobs` und dessen Action `ls` weitergeleitet. Innerhalb einer Controller-Action können HTTP-Parameter abgefragt, Daten aus der Datenbank (bzw. hier aus dem Dateisystem) beschafft und anschließend an den Anfragenden zurückgeliefert werden. Dabei wird das Ausgabeformat abhängig von Dekoratoren umgewandelt.

Das Listing 5.4 zeigt die Methode `ls` aus dem `jobs`-Controller. Der Dekorator `@jsonify` sorgt dafür, dass das zurückgegebene Objekt automatisch in JSON konvertiert wird und der MIME-Type für diese HTTP-Antwort auf `application/json` gesetzt wird.

Als Eingabeparameter dient `node`, welche das zu ladende Verzeichnis bestimmt. Die jeweiligen Dateien werden, zusammen mit den definierten Datenfeldern (size, changed, file), gesammelt und am Ende zurückgegeben.

```
1  class JobsController(BaseController):
2
3   @jsonify
4   def ls(self):
5    node = request.params.get("node", "root")
6    log.debug("ls node %s" % node)
7    x = []
8    if node == "root":
9     x.append({"id": "", "size": 0,
10              "changed": datetime.datetime.now(),
11              "expanded": True, "file": "/"})
12   else:
13    nodepath = os.path.join(config.get("base_path", "/tmp"), node)
14    if os.path.exists(nodepath) and os.path.isdir(nodepath):
15     l = os.listdir(nodepath)
16
```

```
17      for ent in sorted(l):
18       pth = os.path.join(nodepath, ent)
19       stat = os.stat(pth)
20       x.append({ "id": os.path.join(node, ent), "size": stat.st_size,
21           "changed": datetime.datetime.fromtimestamp(stat.st_ctime),
22           "file": ent,
23           "leaf": not os.path.isdir(pth)})
24    count = len(x)
25    success = True
26    return {"ls": x,
27            "total": count,
28            "success": success}
```

Listing 5.4: Pylons-Controller für Dateisystem

Erstellen eines neuen Jobs

Beim Erstellen eines neuen Jobs durch das Frontend müssen zuerst die Daten aus HTTP-Parametern erfasst werden und ein neuer Job in der Datenbank MongoDB (MongoDB, 2012) angelegt werden. Die Dokumentenbasierte Datenbank kann direkt mit Python-Objekten umgehen. Listing 5.5 zeigt mit der Methode `createJobData`, wie ein neuer Job inklusive sämtlicher Metadaten (Plugins, Größen, Unterteilungen) in die Datenbank eingefügt wird. Das Dokument `job` wird dabei in der Sammlung `jobs` mit der Zeile `self.db.jobs.save(job)` abgespeichert.

```
1     def createJobData(self, plugins, fileinfo, title,
2                       desc, concurrency, tasksplit, enqueued):
3         jobid = uuid.uuid4()
4         self.getDB()
5         job = {}
6         job["enqueued"] = enqueued
7         job["jobid"] = str(jobid)
8         job["title"] = title
9         job["start"] = enqueued
10        job["description"] = desc
11        job["tasksplit"] = tasksplit
12        job["concurrency"] = concurrency
13        job["status"] = "enqueued"
14        job["fileinfo"] = fileinfo
15        job["finishedjobs"] = 0
16        job["results"] = []
```

```
17          job["taskids"] = []
18          job["plugins"] = plugins
19          self.db.jobs.save(job)
20          return job
```

Listing 5.5: Dokument in der Datenbank erstellen

Nachdem der Job in der Datenbank abgelegt wurde, wird der eigentliche Analyseprozess gestartet, indem die gerade erzeugte Job-Id als Nachricht an einen Bearbeitungsnode geschickt wird. Das Listing 5.6 zeigt sowohl das Absenden der Nachricht mit einem frühest möglichen Startzeitpunkt (`eta`) und der Job-Id. Die Bearbeitung dieser Nachricht und damit die Aufteilung der Arbeitspakete passiert asynchron, damit das Webinterface sofort wieder verfügbar ist, sobald die Job-Metadaten gespeichert wurden. In diesem Schritt wird der Job als gerade ausführend markiert und die einzelnen Teilaufgaben wiederum als Nachrichten (bestehend aus Job-Id und Block-Index-Nummer) versendet. Die Nachrichten wurden hier bewusst klein gehalten, da der gesamte Zustand der Abarbeitung in der Datenbank gespeichert wird und dadurch eine Verteilung des Wissens vermieden werden soll.

```
1 eta = datetime.datetime.now()
2 jobid = tasks.process_image.apply_async(args=[job["jobid"]], eta=eta
      )
3
4 # Asynchron auf Worker-Node aufgerufen
5 @task(name="process_image")
6 def process_image(jobid):
7     log = process_image.get_logger()
8     log.warning("processing jobid \%r" \% (jobid))
9     PFCDB.setExecuting(jobid)
10    job = PFCDB.getJob(jobid)
11    log.info("Filetype is \%r" \% job["fileinfo"]["filetype"])
12    for i in xrange(job["fileinfo"]["nrjobs"]):
13        PFCDB.appendTask(jobid, process_block.delay(jobid, i))
```

Listing 5.6: Erstellen des Jobs im Queuing-System

Queueing-System zur Plugin-Ausführung

Die erzeugte Beschreibung für einen Carving-Job wird vom Queueing-System bearbeitet. Die einzelnen Beschreibungen werden in der Datenbank MongoDB gespeichert

(MongoDB, 2012). Im Messaging-System RabbitMQ werden jeweils die aktuelle Job-Id und (wenn zutreffend) der zu bearbeitende Block-Index als Nachrichten an die jeweiligen Analyserechner gespeichert und verteilt (RabbitMQ, 2012). Diese Nachrichten werden von der Work-Queue Celery abgearbeitet (Celery, 2012).

Jede Aufgabe besteht aus Teilaufgaben, welche der gewählten Blockgröße entsprechen. Sobald eine Aufgabe vom Analyserechner abgeholt wird, öffnet dieser das Dateisystemabbild und führt die gewählten Plugins für diesen Block aus. Sobald der letzte Block bearbeitet wurde, werden diese Ergebnisse zusammengefasst und können anschließend im Webinterface wieder abgerufen werden.

Listing 5.7 zeigt die Analysephase eines kompletten Blocks. Der Dekorator `@task` erzeugt aus der Funktion `process_block` einen Task, welcher in der Work-Queue Celery verfügbar gemacht wird. Mit einer Nachricht bestehend aus der Job-Id und der Blocknummer kann diese Funktion somit asynchron auf beliebig vielen Analyserechnern aufgerufen werden.
Der erste Schritt ist es, mit Hilfe der Job-Id das Beschreibungsdokument für diesen Job aus der Datenbank zu laden. Anschließend wird der tatsächliche Bereich innerhalb des Dateisystemabbilds berechnet. Für jedes im Job-Objekt definierte Plugin wird anschließend eine Instanz erzeugt und das Plugin mit den jeweiligen Parametern und dem Datenblock aufgerufen. Die Ergebnisse werden in einer Liste gesammelt und am Ende der Funktion gemeinsam in die Datenbank geschrieben. Die Funktion `PFCDB.finalizeJobPart` sorgt dafür, dass, sobald der letzte Teil der Analyse geschrieben worden ist, die Teilergebnisse zusammengefügt und sortiert werden.

```
1
2 @task(name="process_block")
3 def process_block(jobid, blockNum):
4   log = process_block.get_logger()
5   job = PFCDB.getJob(jobid)
6   fileinfo = job["fileinfo"]
7
8   offset = fileinfo["jobsize"] * blockNum
9   clustersize = fileinfo["clustersize"]
10
11  results = []
12  jobplugins = job["plugins"]
13  for plug in jobplugins:
```

```
14      plugId = plug["pluginId"]
15      plugin, version = plugId.split("-")
16      pluginInstance = None
17      for regpl in plugins.plugins:
18        if str(regpl.__name__) == plugin and
19           str(regpl.getVersion()) == version:
20          pluginInstance = regpl
21          break
22      if pluginInstance != None:
23        pl = pluginInstance()
24        pl.setParams(plug["parameter"])
25        if not os.path.exists(fileinfo["filename"]):
26          log.error("File %r not existing!" % (fileinfo["filename"]))
27          break
28        if fileinfo["filetype"] == "ewf":
29          filepointer = ewffile(fileinfo["filename"])
30        elif fileinfo["filetype"] == "dd":
31          filepointer = open(fileinfo["filename"], "rb")
32
33        res = pl.process(filepointer=filepointer, offset=offset,
34                         clustersize=clustersize,
35                         end=offset + fileinfo["jobsize"],
36                         filesize=fileinfo["filesize"])
37        results.append(res.toObject())
38        filepointer.close()
39      else:
40        log.error("Plugin not found: %r" % (plugId))
41    PFCDB.finalizeJobPart(jobid, blockNum, results,
42                          process_block.request.id)
43    log.warning("Block finished: %r %r" % (jobid, blockNum))
```

Listing 5.7: Analyse eines Blocks

Es können beliebig viele dieser Analyserechner gestartet werden, die Verteilung der Aufgaben wird dabei vom Messaging-System übernommen. Fällt ein Rechner aus werden die noch nicht vollständig bearbeiteten Blöcke von anderen Rechnern automatisch übernommen.

5.1.3 Architekturentscheidungen

Die Benutzeroberfläche von Parallelized File Carving ist als Web-Applikation ausgeführt. Der Vorteil von ExtJS als JavaScript-Bibliothek liegt darin, dass keine weiteren Plugins wie Java oder Adobe Flash installiert werden müssen. Die einzige Voraussetzung, um das System bedienen zu können, ist ein graphischer Web-Browser.
Dadurch kann die Installation zentral gewartet werden, da sämtlicher Code auf dem Server liegt und ausgetauscht werden kann. Zusätzlich ist ein Netzwerkzugriff und damit ein Zusammenarbeiten von mehreren Personen möglich.

Das Backend bietet eine JSON-Schnittstelle an, welche vom Frontend genutzt wird. Diese Schnittstelle kann auch von weiteren Programmen weiterverwendet werden, um so zum Beispiel Auswertungen ohne Nutzerinteraktion automatisch starten zu lassen. JSON ist in vielen Programmierumgebungen sehr gut unterstützt und ist auch für den Entwickler ein lesbares Datenformat. Weiters ist der Speicheroverhead im Vergleich zu XML geringer und das Format wird von JavaScript besser unterstützt.

Die Aufteilung der Arbeitspakete auf Prozessebene und die Verteilung auf mehrere Hosts in der Grundarchitektur begründet sich in der Annahme, dass die Festplattenbandbreite ein stark limitierender Faktor beim File Carving ist. Daher ist eine Trennung auf mehrere Analyserechner (mit der Möglichkeit auch die Dateisystemabbilder auf diese zu partitionieren) sinnvoll und bietet weitere Möglichkeiten, den Analyseprozess zu beschleunigen.

5.2 Unterstützte Dateiformate

Parallelized File Carving unterstützt mehrere Typen von Dateisystemabbildern. Darunter befinden sich 1:1-Abbilder auf Byte-Ebene und komplexere Formate wie EWF (Expert Witness Compression Format (Metz, 2012)). Dieser Abschnitt zeigt, wie Abbilder in den jeweiligen Formaten mit Open Source-Werkzeugen erzeugt werden können. Die spezifischen Eigenschaften von diesen Formaten sind im Abschnitt 2.3 erklärt.

5.2.1 Raw-Abbild

Raw-Abbilder können unter Linux mit dem Programm `dd` aus den GNU coreutils (GNU, 2012) erzeugt werden. Dabei wird meistens aus einer physikalischen Festplatte oder einer Partition eine Abbilddatei erzeugt.

Die wichtigsten Parameter sind `if` für die Eingabe und `of` für die Ausgabe. In Listing 5.8 wird die Festplatte `sda` in das Abbild `image.dd` gesichert.

```
1 pfchost:~$ dd if=/dev/sda of=image.dd
```

Listing 5.8: Erzeugen eines Raw-Abbilds mit dd

Basierend auf `dd` gibt es eine erweiterte Version, `dcfldd`, welche Funktionen für den forensischen Einsatz bereitstellt. Dieses Werkzeug wurde ursprünglich von Nicholas Harbour am Department of Defense Computer Forensics Lab entwickelt (Harbour, 2006). Dieses Programm ermöglicht es, während des Auslesevorgangs gleichzeitig Hashwerte zu erstellen oder das Abbild verteilt auf mehrere Zielfestplatten zu schreiben.

Durch die nebenläufige Hashwertbildung wird die Erfassung der Abbilder deutlich beschleunigt, da ansonsten das gesamte Abbild noch einmal eingelesen werden müsste, um den Hashwert zu bilden. Das Listing 5.9 zeigt ein Beispiel, in dem einmal ein Abbild mit `dd` und anschließendem `md5sum` ausgelesen wird und einmal der Hashwert direkt beim Auslesen gebildet wird.

```
1 #Erzeugen eines Abbilds mit dd
2 pfchost:~$ dd if=/dev/zero of=image2.dd bs=1M count=256
3 #Erzeugen des Hashwerts
4 pfchost:~$ md5sum image2.dd
5 1f5039e50bd66b290c56684d8550c6c2  image2.dd
6
7 #Erzeugen eines Abbilds mit Hashwert
8 pfchost:~$ dcfldd if=/dev/zero of=image.dd bs=1M count=256 hash=md5
    hashlog=md5sum_dcfldd.txt
9 pfchost:~$ cat md5sum_dcfldd.txt
10 Total (md5): 1f5039e50bd66b290c56684d8550c6c2
```

Listing 5.9: Erzeugen eines Raw-Abbilds mit automatischer Hashwert-Erzeugung

Auf einem Testsystem (Lenovo Thinkpad 420s mit 4GB RAM, 128GB Intel-SSD) beliefen sich die Zeiten für ein 2GB-Abbild auf 27s (`dd` und `md5sum`) gegenüber 22s (`dcfldd`).

5.2.2 EWF-Abbild

Das Expert Witness Compression Format ist ein proprietäres Format, welches beim Produkt EnCase Forensics eingesetzt wird. Beim Auslesen eines Abbilds mit den Programm `ewfacquire` (als Teil der libewf (Metz, n.d.)) werden Metadaten zu dem Abbild verknüpft.

Diese Metadaten enthalten eine Fallnummer, eine Beweisnummer, den Namen der ausführenden Person oder eine Beschreibung. Das Programm bildet automatisch, wie `dcfldd`, einen Hashwert während des Auslesens. Das Abbild wird auch automatisch auf mehrere Dateien aufgeteilt, die Standardteilung liegt bei 1.4 GiB pro Teilstück. Zusätzlich kann das Abbild auch on-the-fly verlustlos komprimiert werden, um Speicherplatz zu sparen.

Listing 5.10 zeigt das Erzeugen eines EWF-Abbilds von der physikalischen Festplatte `sda`. Der Parameter `-u` bewirkt, dass `ewfacquire` sofort mit dem Auslesen beginnt. Fehlt dieser Parameter, muss der Benutzer sämtliche Metadaten des Abbilds eingeben. Das Abbild wird im aktuellen Verzeichnis unter dem Namen `ewfimage.E01` abgelegt. Ist das Abbild größer als die Teilung, werden Dateien mit einem aufsteigenden Index, also `ewfimage.E02`, angelegt.

```
1  pfchost:~$ ewfacquire /dev/sda -t ewfimage -u
2  ewfacquire 20120504
3
4  Storage media information:
5  Type:                 RAW image
6  Media size:             2.1 GB (2147483648 bytes)
7  Bytes per sector:       512
8
9  Acquiry started at: Fri Aug 24 09:54:44 2012
10
11 This could take a while.
12 Status: at 0%.
13         acquired 32 KiB (32768 bytes) of total 2.0 GiB (2147483648
               bytes).
14 [...]
15
16 Acquiry completed at: Fri Aug 24 09:55:21 2012
17
```

```
18 Written: 2.0 GiB (2147483836 bytes) in 37 second(s) with 55 MiB/s
       (58040103 bytes/second).
19 MD5 hash calculated over data:        a981130cf2b7e09f4686dc273cf7187e
20 ewfacquire: SUCCESS
```

Listing 5.10: Erzeugen eines EWF-Abbilds

5.2.3 AFF-Abbild

Das Advanced Forensics Format wurde von Garfinkel et. al. (Garfinkel, 2006) entwickelt, um ein offenes Format mit ähnlichem Funktionsumfang wie EWF zur Verfügung zu stellen. Um eine Festplatte oder ein Festplattenabbild in das AFF-Format zu konvertieren, wird das Programm `affconvert` (als Teil der afflib (Garfinkel, 2006)) verwendet.

Beim Auslesen der Daten werden ähnlich wie bei den vorherigen Dateiformaten Hashwerte gebildet. Zusätzlich zum MD5-Hash wird auch ein SHA-1-Hash abgespeichert. Werden keine weiteren Parameter gesetzt, wird das erzeugte Abbild automatisch komprimiert.

Listing 5.11 zeigt den Ablauf, um mit `affconvert` ein Dateisystemabbild zu erzeugen. Zusätzlich soll das Ergebnis noch in 4 GB große Teile abgespeichert werden, um die Archivierung zu vereinfachen. Dazu wird der Parameter `-M4g` gesetzt.

```
1 pfchost:~$ affconvert -M4g /dev/sda -o image.aff
2 convert /dev/sda --> image.aff
3 Converting page 0 of 127
4 [...]
5 md5: d4a160a38202b682ace69afbc6ccd942
6 sha1: 8f43cb38c4e94f60b6bfd6ddd8caa06e81674e3c
7 bytes converted: 2147483648
8 Total pages: 128  (128 compressed)
9 Conversion finished.
```

Listing 5.11: Erzeugen eines AFF-Abbilds

5.3 Verteilung auf mehrere Hosts

Der Grad der Parallelisierung wird bei Parallelized File Carving von der Anzahl der Prozessoren in einem Rechenknoten und der Anzahl der Rechenknoten bestimmt. Die Verteilung der Carving-Verarbeitung wird in diesem Abschnitt erklärt.

5.3.1 Zugriff auf die Abbilder

Die forensisch erzeugten Dateisystemabbilder müssen in einem Rechnerverbund allen teilnehmenden Systemen zur Verfügung stehen. Dies wird durch die Verwendung eines Netzwerkdateisystems, in diesem Fall NFS, garantiert.

Das Abbild wird auf einem Storage-Server abgelegt und dann den einzelnen Rechenknoten (und auch dem Master-Knoten, welcher die Aufgaben erzeugt) zugänglich gemacht. Dieser Storage-Server muss die Daten ausreichend schnell zur Verfügung stellen. Die Geschwindigkeit kann durch den Einsatz schnellerer Festplatten, die Erhöhung der Spindelanzahl im RAID oder dem Einsatz von RAM- oder SSD-Caches verbessert werden.

Weitere Möglichkeiten zur I/O-Skalierung werden im Kapitel 9 skizziert, indem GlusterFS auf den einzelnen Rechenknoten zum Einsatz kommt.

Das verwendete Netzwerkdateisystem muss dabei eine Unterstützung für wahlfreien Zugriff auf die Dateien bieten, da die einzelnen Rechnerknoten nur auf Teilstücke der Gesamtabbilder zugreifen und nicht die gesamten Dateien einlesen.

Damit alle Rechenknoten mit einem gleichen Namen auf das Abbild zugreifen können, empfiehlt sich ein Einbinden unter dem selben Verzeichnis, z.B. `/home/fc`.

5.3.2 Organisation der Aufgaben

Auf dem Master-System wird die Datei in Arbeitsaufgaben einer bestimmten Größe unterteilt. Diese Aufgaben werden in eine verteilte Messaging-Queue (RabbitMQ, siehe Abschnitt 6.1 zur Installation) abgelegt.

Die Arbeitspakete bestehen aus folgenden Teilen:

- Der Dateiname des Abbilds, welches sich im Netzwerkdateisystem befindet, um die Datei öffnen und lesen zu können.

- Die Grenzen dieses Arbeitspakets, definiert durch die Start-Position und die Länge.
- Eine Beschreibung der durchzuführenden Aufgaben. Im Wesentlichen wird dazu eine Plugin-Liste, gekoppelt mit den dazugehörigen Parametern, angehängt.

Jeder Rechenknoten verbindet sich zu dieser Queue und fordert jeweils so viele Aufgaben an, wie Prozessoren in diesem System vorhanden sind. Die Teilergebnisse werden anschließend von jedem System einzeln in eine zentrale Datenbank abgelegt.

Die Grenzen von den Arbeitspaketen dürfen in Ausnahmefällen übertreten werden. Ein Beispiel dafür ist, dass ein Rechner in den letzten Kilobytes seines Pakets den Anfang einer Datei gefunden hat. Dieser Rechner darf in dem Fall dann noch bis zur maximalen Dateigröße im Abbild weiterlesen, um einen aufwändigen zweiten Durchlauf zu vermeiden.

5.3.3 Abschluss der Analyse

Sobald alle Ergebnisse der einzelnen Arbeitspakete vorliegen, werden diese aufbereitet. Dabei werden alle Fragmente, welche in beliebiger Reihenfolge gefunden werden konnten, sortiert und zusammengefügt. Der Benutzer kann anschließend die Ergebnisse betrachten und findet im Vergleich zur Analyse auf einem System das selbe Ergebnis vor.

5.4 Beispielplugin: PNG/JPEG-Erkennung

Das Beispielplugin zur PNG- und JPEG-Erkennung in Dateisystemabbildern dient dazu, den gesamten Funktionsumfang von Parallelized File Carving zu demonstrieren. Zusätzlich dient dieses Plugin als Vorlage für Plugins, welche Parallelized File Carving später erweitern werden.

Das Plugin kann durch drei Parameter gesteuert werden: die maximale Dateigröße `maxfilesize` und jeweils ob nach PNG- oder JPEG-Dateien gesucht werden soll.

5.4.1 PNG-Erkennung

Abschnitt 2.7 auf Seite 16 zeigt die grundlegende Erkennungsroutine für PNG-Dateien innerhalb eines Dateisystemabbilds. Die PNG-Erkennungsroutine verwendet die Struk-

tur des PNG-Formats, indem Block für Block durchsucht wird, bis der letzte, speziell gekennzeichnete, Block eines PNG-Bildes gefunden wurde.

5.4.2 JPEG-Erkennung

Das JPEG-Plugin verwendet einen einfacheren Ansatz. Das Plugin hat nur einen einzelnen Heuristik-Parameter, nämlich die maximale Dateigröße. Um zu entscheiden, ob ein JPEG-Bild gefunden wurde, werden folgende zwei Kriterien hinzugezogen: das Bild beginnt mit Hex `0xFFD8` und endet mit `0xFFD9`.
Dieses Plugin liefert einige False-Positives, zeigt jedoch auch, wie mit geringem Aufwand ein neuer Dateityp als Plugin zu Parallelized File Carving hinzugefügt werden kann.

Listing 5.12 zeigt die Erkennungsroutine, sobald ein Anfang gefunden wurde (Muster `0xFFD8`), wird solange gelesen, bis entweder das End-Muster (`0xFFD9`) oder die maximale Dateigröße erreicht wurde.
Dieses Beispiel nimmt an, dass der Beginn der Datei an einer Sektorgrenze liegt und die Gesamtdatei eine gerade Anzahl von Bytes aufweist. Für eine korrekte Implementierung müssten auch ungerade Grenzen betrachtet werden.

```
1  def _checkJPG(f, off):
2    inoff = off
3    maxoff = off + MAX_SIZE
4    f.seek(off)
5    data = f.read(2)
6    off += 2
7    if data == "\xFF\xD8":
8      block = ""
9      while block != "\xFF\xD9" and off < maxoff:
10       block = f.read(2)
11       off += 2
12     if off >= maxoff:
13       return inoff
14     else:
15       return off
16   else:
17     return inoff
```

Listing 5.12: Erkennen einer JPEG-Datei

5.5 Vorschau

Beim File Carving werden häufig sehr viele Dateien gefunden, welche sich bei der späteren Untersuchung als nicht benötigt herausstellen. Klassische File Carving-Programme extrahieren jede Datei aus dem Abbild, was im schlimmsten Fall zu einem vielfachen Speicherplatzbedarf führt, da sowohl das Abbild als auch die gefundenen Dateien abgelegt werden.

Die Inspektion der Dateien ist jedoch mit den Koordinaten (an welchen Byte-Offsets befindet sich die Datei innerhalb des Abbilds) möglich, indem die Datei on-the-fly aus dem Abbild ausgelesen wird.

Die Bibliothek `CarvFS` bietet ein Linux Filesystem in Userspace (FUSE) an. Dadurch kann auf Teile des Abbilds über normale Dateisystemoperationen zugegriffen werden. CarvFS ist Teil der Open Computer Forensics Architecture und wurde von der niederländischen Polizei entwickelt. Zu den unterstützten Abbildtypen zählen Raw-Abbilder (1:1), EWF-Abbilder (Expert Witness Compression Format) und AFF (Advanced Forensics Format) (CarvFS, 2010).

5.5.1 Verwendung von CarvFS

Um ein Abbild mit CarvFS bearbeiten zu können, muss das Abbild mit einem Verzeichnis verknüpft werden (mounten). In diesem Verzeichnis kann dann auf virtuelle Dateien, welche Bereiche im Abbild repräsentieren, zugegriffen werden.

Das Listing 5.13 bindet eine Abbilddatei ein und greift über virtuelle Dateien auf Dateien innerhalb des Abbilds zu. Eine wichtige Einschränkung ist, dass die Dateien nicht aufgelistet werden können, da CarvFS selbst keine Dateioffsets verwaltet. Dadurch liefert das Kommando `ls -l` auch keine Ergebnisse.

Der Zugriff auf die Datei `0+512.crv` liefert ab dem Offset 0 beginnend 512 Bytes. Bei vollständigen Festplattenabbildern findet man an dieser Stelle den Master Boot Record (bei nicht GPT-partitionierten Festplatten), in diesem Beispiel direkt den Dateisystem-Header einer FAT16-Partition.

Der Zugriff erfolgt on-the-fly, das heißt für den Zugriff wird die Datei nicht extrahiert und temporär abgespeichert, sondern direkt aus dem Abbild in der richtigen Reihen-

folge durchgereicht.

```
1 pfchost:~$ carvfs /home/fc/carvfs raw carvmount abbild.dd
2 pfchost:~/carvfs/carvmount/CarvFS$ ls -l
3  total 0
4 pfchost:~/carvfs/carvmount/CarvFS$ hd 0+512.crv
5 eb 3c 90 6d 6b 64 6f 73 66 73 00 00 02 10 01 00 |.<.mkdosfs......|
6 02 00 02 00 00 f8 00 01 20 00 40 00 00 00 00 00 |........ .@.....|
7 00 00 10 00 00 00 29 d1 36 bd 7d 56 46 41 54 54 |......).6.}VFATT|
8 45 53 54 20 20 20 46 41 54 31 36 20 20 20 0e 1f |EST FAT16   ..|
9 [...]
```

Listing 5.13: Verwendung von CarvFS

Besteht die Datei aus mehreren, nicht zusammenhängenden Fragmenten, kann der Dateiname auch eine Liste von Fragmenten spezifizieren. Man fügt dazu mehrere Paare aus Offset und Länge zusammen. Eine Datei mit zwei Fragmenten, beginnend bei 1024 mit einer Länge von 4096 Bytes und beginnend bei 16384 mit 4096 Bytes, wird als `1024+4096_16384+4096.crv` angesprochen.

Dateien, welche stark fragmentiert sind, führen durch diesen Ansatz zu sehr langen Dateinamen. Die maximale Länge eines einzelnen Dateinamen ist unter Linux auf 255 Zeichen beschränkt, in diesem Fall müssen die Fragmente nacheinander ausgewertet werden oder der Zugriff erfolgt direkt über `libcarvpath` ohne das virtuelle File-System zu verwenden.

5.5.2 Zugriff auf Dateien in Parallelized File Carving

Sobald die Analyse eines Abbilds fertiggestellt ist, können die einzelnen Dateien betrachtet und extrahiert werden. Die von den Plugins gelieferten Koordinaten (Fragmentlisten mit Start und Länge der jeweiligen Blöcke) werden dazu transparent in virtuelle Dateien übersetzt.

Der Zugriff auf die Dateien erfolgt über CarvFS. Dazu wird beim Zugriff auf eine Datei automatisch das Abbild über CarvFS eingebunden (gemountet) und symbolische Links auf die gefundenen Dateien erzeugt. Die so referenzierten Dateien werden direkt an den Webbrowser des Benutzers ausgeliefert.

Der genaue Ablauf für eine PNG-Datei bestehend aus zwei Fragmenten wird in Abbildung 5.1 dargestellt. Der Webbrowser lädt das Bild über die URL `/results/getFile?id=18` und zeigt das PNG-Bild mit einem img-Tag direkt an.

Sobald eine gewisse Zeit kein Zugriff auf das Abbild erfolgt, wird dieser CarvFS-Mount automatisch wieder entfernt, um unnötigen Ressourcenverbrauch zu vermeiden.

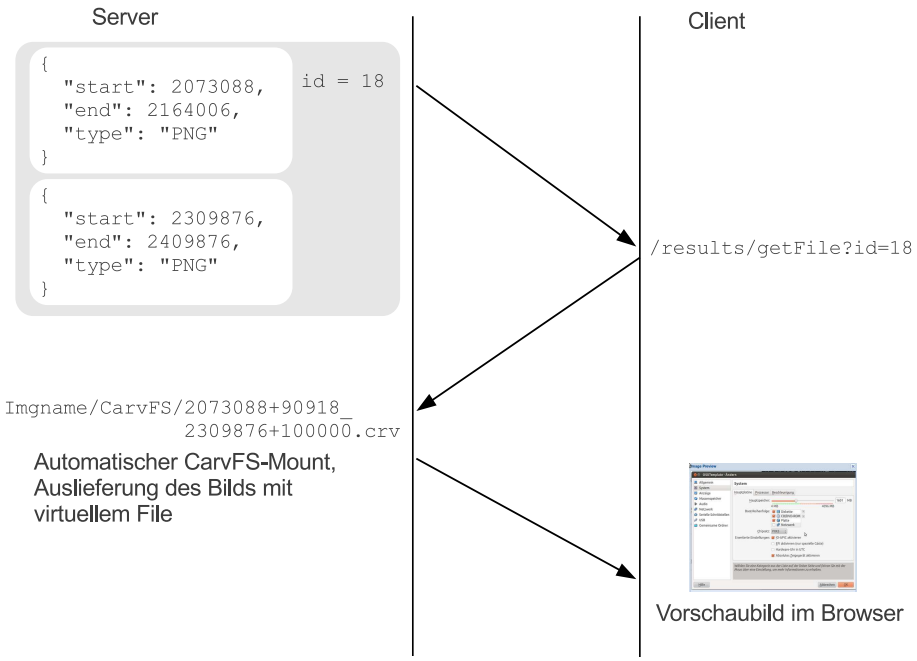


Abbildung 5.1: Verwendung von CarvFS in Parallelized File Carving

5.6 Geschwindigkeit / Speed-Up

Ein Ziel von Parallelized File Carving ist es, die Geschwindigkeit der Dateisystemabbild-Analyse zu beschleunigen. Damit das Ergebnis validiert werden kann, wurden ausführliche Benchmarks durchgeführt.

Die Maßzahl, welche die Geschwindigkeitssteigerung bezeichnet, wird als Speedup S_p bezeichnet, wobei T_1 die Ausführungszeit auf einem einzelnen Prozessor bezeichnet und T_p die Ausführungszeit auf p Prozessoren angibt (siehe Formel 5.1).

$$S_p = \frac{T_1}{T_p} \tag{5.1}$$

Der ideale Speedup liegt bei $S_p = p$, das bedeutet, jeder hinzugefügte Prozessor beschleunigt die Ausführung linear.
Die Effizienz (siehe Formel 5.2) bezeichnet im Wertebereich von 0 bis 1, wie gut dieser optimale Speedup erreicht wurde.

$$E_p = \frac{S_p}{p} \tag{5.2}$$

Dieser Abschnitt zeigt, wie die Testdaten und die Testumgebung aufgebaut sind. Die Testläufe bearbeiten das gleiche Abbild mit steigender Zahl der parallel darauf zugreifenden Analyseprozesse. Im letzten Teil dieses Abschnittes werden die erhaltenen Ergebnisse dargestellt und diskutiert.

5.6.1 Erzeugen der Testdaten

Grundlage des Geschwindigkeitsvergleichs ist ein Testabbild im FAT32-Format. Das Image ist 5GB groß und mit 20.000 identen PNG-Bildern mit jeweils 200 kB Größe gefüllt.
Dieses Abbild wurde wie in Listing 5.14 gezeigt erzeugt.

```
1 pfchost:/mnt/Benchmark$ dd if=/dev/zero of=testfile_5g.dd bs=1M
    count=5120
2 5120+0 Datensaetze ein
3 5120+0 Datensaetze aus
4 5368709120 Bytes (5,4 GB) kopiert, 41,2025 s, 130 MB/s
5 pfchost:/mnt/Benchmark$ mkfs.vfat testfile_5g.dd
6 mkfs.vfat 3.0.7 (24 Dec 2009)
7 pfchost:/mnt/Benchmark$ ls -lA /tmp/testimage.png
8 -rw-rw-r-- 1 gregi gregi 203108 2012-09-16 18:33 /tmp/testimage.png
9 pfchost:/mnt/Benchmark$ mount -o loop testfile_5g.dd loop/
```

```
10 pfchost:/mnt/Benchmark$ for i in {1..20000}; do
11     cp /tmp/testimage.png loop/image$i.png;
12     done
13 pfchost:/mnt/Benchmark$ umount loop
```

Listing 5.14: Erzeugen des Test-Images

In diesem Beispiel wurden bewusst keine Dateien gelöscht, um die Auswertedauer nicht durch Fragmente zu beeinflussen und damit den Speed-Up der Parallelisierung genau zu betrachten.

5.6.2 Testaufbau

Der Testaufbau für die Geschwindigkeitstests besteht aus einer virtuellen Maschine mit 1024 MB RAM und zwei CPU-Cores eines Intel(R) Core(TM) i5-2520M Prozessors bei 2.50GHz Taktrate. Das verwendete Speichermedium ist eine Intel SSDSA2BW120G3 Solid State Disk mit 120 GB. Das Host-Betriebssystem ist Ubuntu 12.04.1 64-Bit, als Virtualisierungssoftware wird Oracle VirtualBox 4.1.22 verwendet. Das Gastbetriebssystem ist Ubuntu 10.04.4 64-Bit.

Das im vorherigen Abschnitt erzeugte Abbild ist direkt in der virtuellen Maschine abgespeichert, um Netzwerklatenzen nicht zu berücksichtigen. Die Tests sind in folgende Kategorien aufgeteilt:

- Task-Größe 128 MB (das bedeutet bei 5GB 40 Einzeltasks) mit PNG-Erkennung
- Task-Größe 64 MB (bei 5GB 80 Einzeltasks) mit PNG-Erkennung
- Task-Größe 128 MB mit PNG-Erkennung und Warte-Plugin (1 Sekunde)
- Task-Größe 64 MB mit PNG-Erkennung und Warte-Plugin (1 Sekunde)

Jede dieser vier Testkategorien ist für die Analyse jeweils mit einem bis vier Bearbeitungsprozessen auf einem Rechner abgelaufen.

5.6.3 Testergebnisse und Diskussion

Laufzeiten PNG-Plugin

Die folgende Tabelle 5.1 zeigt die detaillierten Testergebnisse für die Durchläufe des PNG-Plugins mit einer Task-Größe von 128 MB und 64 MB bei einer Parallelisierung von einem bis vier Prozessen.

Die Zeit *Prepare* bezeichnet die Zeit, welche vom Start des Jobs bis zum Beginn des ersten Arbeitstasks benötigt wird. Innerhalb dieser Zeit wird das Abbild in Einzeltasks aufgeteilt und diese gestartet. In der Zeit *Runtime* wird das ausgewählte Plugin (PNG-Erkennung) ausgeführt.

Totaltime ist die Summe der beiden Zeiten und wird auch für die nachfolgende Speedup-Berechnung herangezogen.

Lauf	**Prepare** [s]	**Runtime** [s]	**Totaltime** [s]
1 CPU 128 MB PNG	3,4	107,4	110,8
2 CPU 128 MB PNG	5,1	77,5	82,6
3 CPU 128 MB PNG	7,1	60,3	67,4
4 CPU 128 MB PNG	8,7	52,5	61,2
1 CPU 64 MB PNG	2,5	111,6	114,1
2 CPU 64 MB PNG	1,4	78,0	79,4
3 CPU 64 MB PNG	3,3	64,3	67,6
4 CPU 64 MB PNG	6,1	60,7	66,8

Tabelle 5.1: Messwerte der Versuchsreihen PNG-Plugin

Laufzeiten PNG-Plugin + Warte-Plugin

Die nächste Testreihe in Tabelle 5.2 beinhaltet zusätzlich zum PNG-Plugin auch noch ein Warteplugin, welches eine Sekunde wartet. Dieses Warteplugin ist stellvertretend für Plugins, welche auf externe Quellen zugreifen und keine lokale Rechenlast verursachen. Möglich sind hier Abfragen auf Hash-Datenbanken zum Abgleich bekannter Dateien, das Benutzen einer Suchmaschine oder das Auflösen einer IP-Adresse mittels DNS.

Lauf	**Prepare** [s]	**Runtime** [s]	**Totaltime** [s]
1 CPU 128 MB PNG+Wait	4,9	145,9	150,8
2 CPU 128 MB PNG+Wait	5,4	89,7	95,1
3 CPU 128 MB PNG+Wait	7,1	66,8	73,9
4 CPU 128 MB PNG+Wait	10,2	56,9	67,1
1 CPU 64 MB PNG+Wait	3,2	190,4	193,6
2 CPU 64 MB PNG+Wait	3,0	105,8	108,8
3 CPU 64 MB PNG+Wait	4,3	83,5	87,8
4 CPU 64 MB PNG+Wait	5,4	78,2	83,6

Tabelle 5.2: Messwerte der Versuchsreihen PNG-Plugin und Warteplugin

Beide Messreihen sind in der Abbildung 5.2 graphisch dargestellt. Es ist zu beobachten, dass der Speedup ab dem zweiten Prozessor nicht mehr deutlich zunimmt. Das ist dadurch begründet, dass die verwendete CPU nur vier virtuelle Kerne (zwei echte Kerne verdoppelt durch Hyper-Threading) besitzt.

Speedup und Effizienz

Die Tabelle 5.3 zeigt den berechneten Speedup in Abhängigkeit der Anzahl der verwendeten CPUs. In der Abbildung 5.3 ist ebenfalls ersichtlich, dass der ideale Speedup bei mehr als zwei CPU-Kernen nicht erreicht werden kann.

Bei den Messreihen, welche das Warteplugin beinhalten, steigt der Speedup auch bei weiteren CPUs noch an. Speziell komplexe externe Auswertungen können von dieser Beschleunigung profitieren, da die Bearbeitung dann nicht so stark von diesen externen Diensten verlangsamt wird.

Die Größe der Tasks hat in diesen Messreihen keinen signifikanten Einfluss auf die

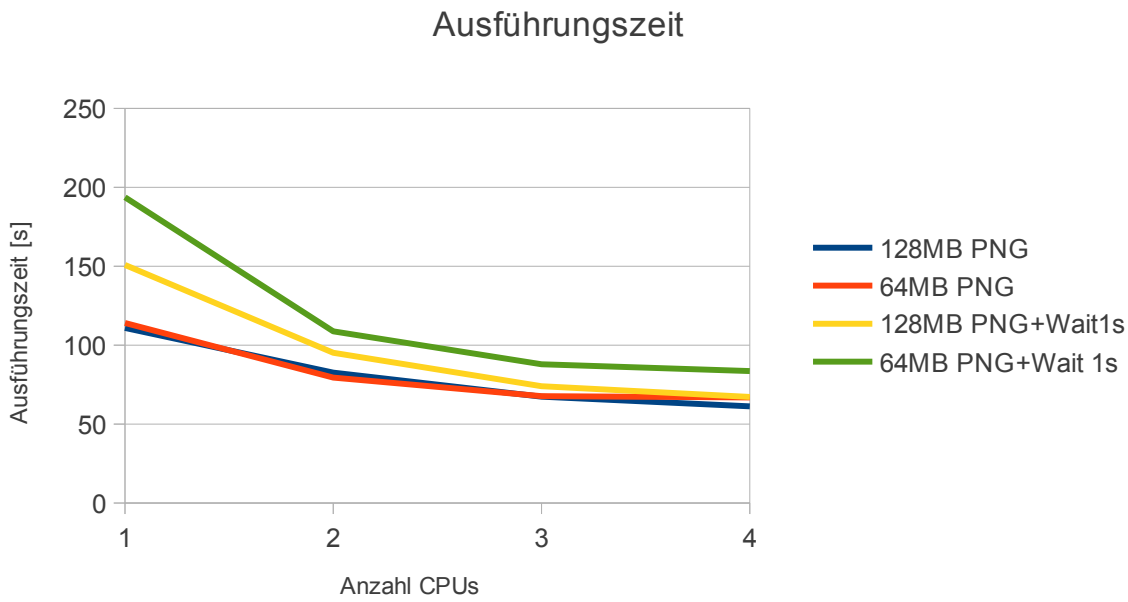


Abbildung 5.2: Laufzeit abhängig von Taskgröße und CPU-Anzahl

Ausführungszeit, sowohl 64 MB-Arbeitsblöcke als auch 128 MB-Arbeitsblöcke liefern ähnliche Ausführungszeiten. Die Vermutung liegt darin, dass die Bearbeitungsgeschwindigkeit in diesem Fall von der verwendeten Festplatte beschränkt wurde.

Lauf	**2 CPU** [1]	**3 CPU** [1]	**4 CPU** [1]
128MB PNG	1,34	1,64	1,64
64MB PNG	1,44	1,69	1,71
128MB PNG+Wait 1s	1,59	2,04	2,25
64MB PNG+Wait 1s	1,78	2,21	2,32

Tabelle 5.3: Speedup der Parallelisierung

In der Tabelle 5.4 wird die Effizienz der Parallelisierung dargestellt. Auch hier ist die Beibehaltung der Effizienz in den Fällen, in denen das Warte-Plugin verwendet wurde, zu beobachten. Abbildung 5.4 zeigt wieder die erfassten Werte im Vergleich zur idealen Effizienz.

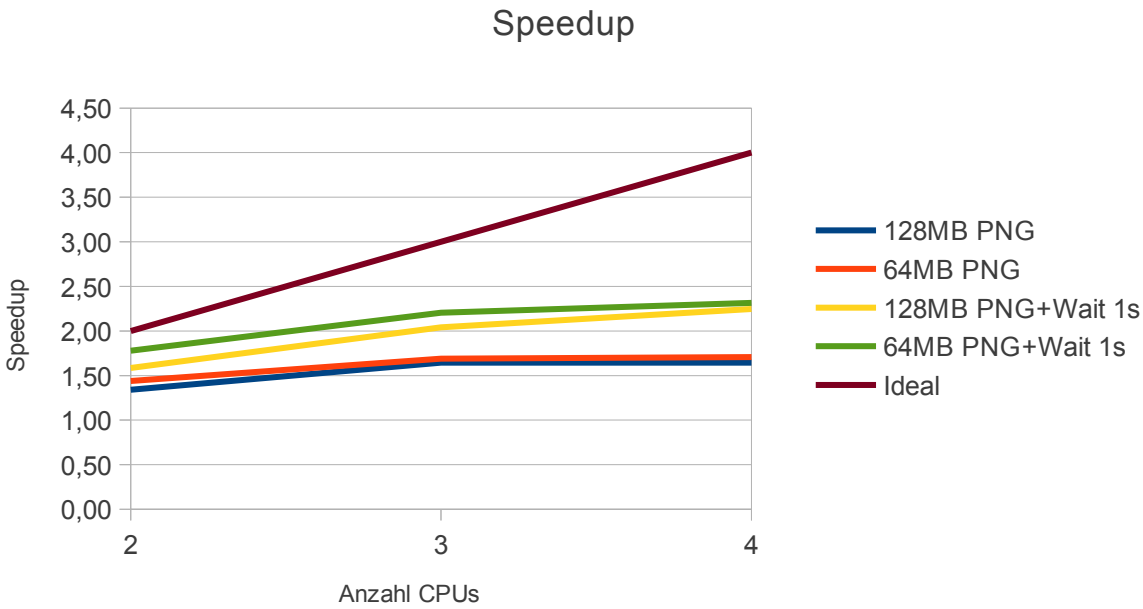


Abbildung 5.3: Speedup abhängig von Taskgröße und CPU-Anzahl

Lauf	**2 CPU** [1]	**3 CPU** [1]	**4 CPU** [1]
128MB PNG	0,67	0,55	0,41
64MB PNG	0,72	0,56	0,43
128MB PNG+Wait 1s	0,79	0,68	0,56
64MB PNG+Wait 1s	0,89	0,74	0,58

Tabelle 5.4: Effizienz der Parallelisierung

Analyse

Die vorgestellten Testreihen wurden bewusst so gewählt, dass auch Grenzfälle in der Parallelisierung aufgezeigt werden.

Größere Installationen sollten auf jeden Fall schnelle Plattensysteme mit einer hohen Anzahl von IOPS (Ein-Ausgabeoperationen pro Sekunde) zur Verfügung stellen, um den Speedup nicht durch die Geschwindigkeit der Festplatten zu begrenzen. Die durch-

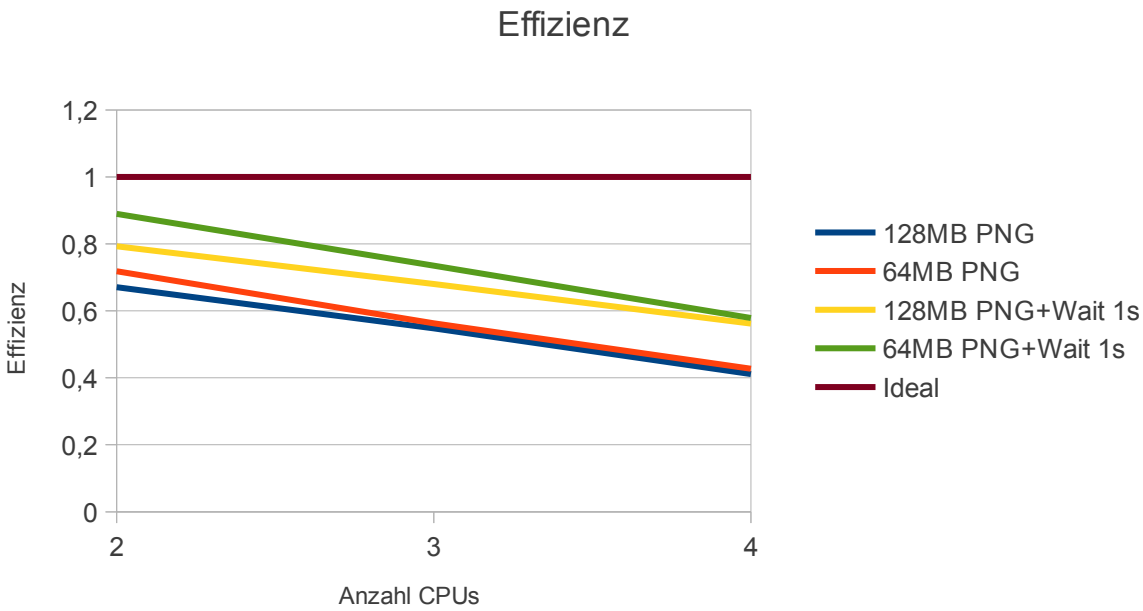


Abbildung 5.4: Effizienz abhängig von Taskgröße und CPU-Anzahl

geführten Testreihen auf einer SSD-Festplatte sind in der Praxis nicht in diesem Ausmaß durchführbar, da die zu analysierenden Datenmengen die Kapazität von gebräuchlichen SSDs übersteigen. Der Einsatz von SSD-Lese-Caches im Plattensystem ermöglicht es, auch mit geringerer SSD-Kapazität von der Geschwindigkeit dieser zu profitieren.

Ein heterogener Plugin-Mix (also abwechselnd Plugins welche die CPU oder die Festplatte belasten und Plugins, welche externe Services verwenden) sorgt für den besten Speedup, da hier Hardware-Ressourcen am besten ausgelastet werden können. Die Messreihen mit dem Warte-Plugin skizzieren diese Mischung von Plugins.

Abschließend gilt es zu erwähnen, dass die Beschleunigung der Analyse in diesen Messreihen zufriedenstellend verlaufen ist und mit dem Einsatz von Parallelized File Carving auf verteilten Nodes zusätzliche Beschleunigung zu erwarten ist.

6 Benutzerhandbuch

Um Parallelized File Carving zu betreiben, ist zumindest ein Linux-Server notwendig. Neben Ubuntu 10.04 LTS wird CentOS 6.2 als Distribution unterstützt.
Dieser Server kann sowohl die Benutzerschnittstelle, als auch das eigentliche Carving vornehmen. Bei größeren Analysen ist es jedoch sinnvoll, mehrere Rechner zu einem Carving-Cluster zu verbinden, um die Auswertung zu beschleunigen.
Die Hardware-Auswahl ist nicht Teil dieses Abschnittes. Ein ausreichend schnelles und verfügbares Storage-System wird jedoch vorausgesetzt (SAN, NAS).

Der Anwender muss keine weitere Software auf seinem PC installieren, um Parallelized File Carving zu verwenden. Die Benutzeroberfläche wird in einem Web-Browser (unterstützt sind Google Chrome ab Version 18, Mozilla Firefox ab Version 12 und Microsoft Internet Explorer ab Version 8) dargestellt und benötigt keine weiteren Plugins wie Java oder Adobe Flash.

6.1 Installation der Betriebssystemabhängigkeiten

Parallelized File Carving hat folgende Betriebssystem-spezifischen Abhängigkeiten:

- `RabbitMQ` - ein AMQP-kompatibler Message Broker, benötigt für die Kommunikation und Aufteilung der Jobs zwischen den Carving-Rechnern (RabbitMQ, 2012), (AMQP, 2012).
- `MongoDB` - eine dokumentenbasierte NoSQL-Datenbank, speichert die Ergebnisse der Carving-Vorgänge (MongoDB, 2012), (Strozzi, 1998).
- `CarvFS / libcarvpath` - eine Bibliothek zum Zugriff auf Dateien in einem Dateisystemabbild (CarvFS, 2010).

Folgende Python-Bibliotheken werden benötigt:

- `celery` - eine verteilte Task-Queue (auf Basis von RabbitMQ, (Celery, 2012))
- `Pylons` - ein Python-Web-Framework (Pylons, 2010).

- `pymongo` - eine Python-Bibliothek zum Zugriff auf MongoDB (pymongo, 2012).

6.1.1 Installation unter Ubuntu 10.04 LTS

Nachdem das Betriebssystem installiert und sämtliche Updates eingespielt wurden, werden die Betriebssystemabhängigkeiten von Parallelized File Carving installiert.

Das Archiv PFC-1.0.tar.gz auf dem beiliegenden Datenträger muss dazu entpackt werden. Danach werden die Installationsskripte innerhalb des Archivs wie folgt aufgerufen:

```
1 pfchost:~$ sudo ./install_dependencies.sh
2 pfchost:~$ ./create_virtualenv.sh
```

Listing 6.1: Installation der Abhängigkeiten unter Ubuntu 10.04 LTS

Das Skript `install_dependencies.sh` installiert die Betriebssystemabhängigkeiten, das zweite Skript `create_virtualenv.sh` installiert die notwendigen Python-Bibliotheken. Eine Liste der Abhängigkeiten ist in Abschnitt 6.1 gesammelt.

6.1.2 Installation unter CentOS 6.2

Nachdem das Betriebssystem installiert und sämtliche Updates eingespielt wurden, werden die Betriebssystemabhängigkeiten von Parallelized File Carving installiert.

Das Archiv PFC-1.0.tar.gz auf dem beiliegenden Datenträger muss dazu entpackt werden. Danach werden die Installationsskripte innerhalb des Archivs wie folgt aufgerufen:

```
1 pfchost:~$ sudo ./install_dependencies_centos.sh
2 pfchost:~$ ./create_virtualenv.sh
```

Listing 6.2: Installation der Abhängigkeiten unter CentOS 6.2

6.1.3 Starten der Anwendung

Nach der Installation muss sowohl ein Carving-Node (`celeryd`) und das Webinterface (*Pylons*) folgendermaßen gestartet werden:

```
1 pfchost:~$ source virtualenv/bin/activate
2 pfchost:~$ cd src
3 pfchost:~$ ./startcelery.sh &
4 pfchost:~$ cd pfcweb
```

```
5 pfchost:~$ ./start.sh
```

Listing 6.3: Starten der Anwendung

6.2 Bedienung

Parallelized File Carving wird im Web-Browser bedient. Dazu wird die Adresse des Carving-Servers geöffnet, um das Carving-Web-Interface zu starten.

6.2.1 Übersicht

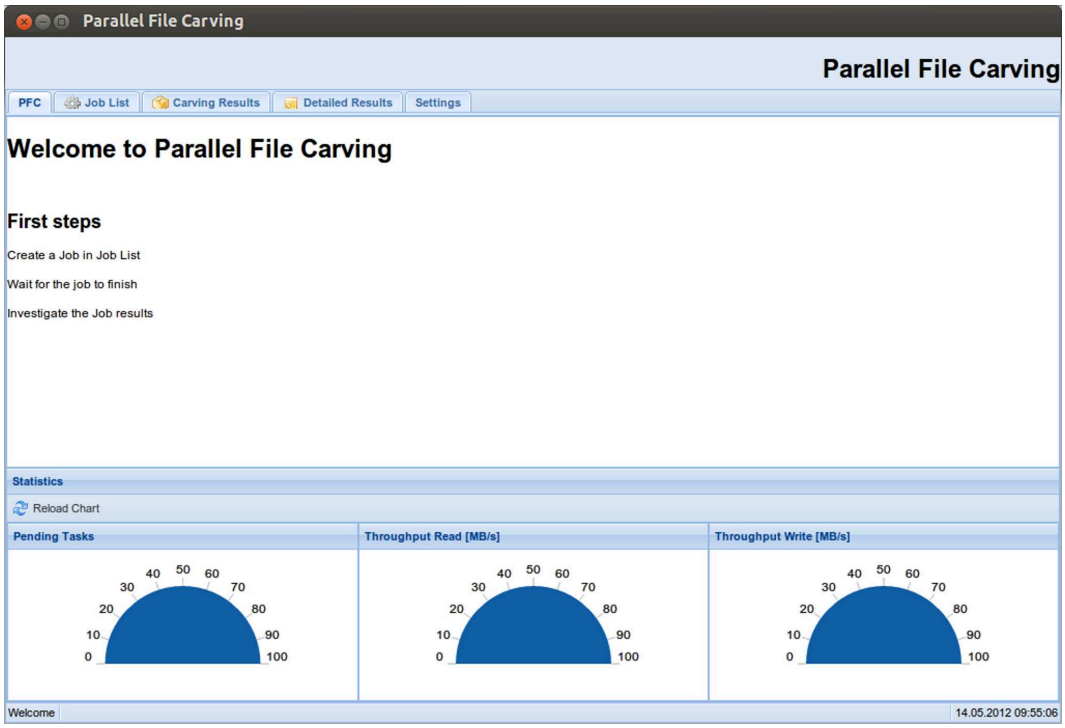
Die Oberfläche von Parallelized File Carving (siehe Abbildung 6.1(a)) strukturiert sich in folgende Teile:

- Im ersten Tab **Home** (Abbildung 6.1(a)) wird ein kurzer Hilfetext dargestellt. Im unteren Bereich findet der Benutzer Informationen zur aktuellen Auslastung, dazu gehören die Anzahl der ausständigen Jobs und die aktuelle Lese- und Schreibrate.
- Unter **Job List** (Abbildung 6.1(b)) können neue Jobs gestartet und bereits laufende oder abgeschlossene Jobs wieder geöffnet werden. Die Liste stellt die Jobs dar, im oberen Bereich können neue Jobs erzeugt, gelöscht, gestartet und angehalten werden.
- Bereits analysierte Jobs werden unter **Carving Results** angezeigt. Nachdem ein Job durch die Dropdown-Auswahl im oberen Bereich ausgewählt wurde, werden die gefundenen Dateifragmente in der Liste dargestellt.
- Genauere Analyse auf Blockebene findet man unter **Detailed Results**. Die gefundenen Dateifragmente werden hier in einer von Hex-Editoren bekannten Darstellung präsentiert.

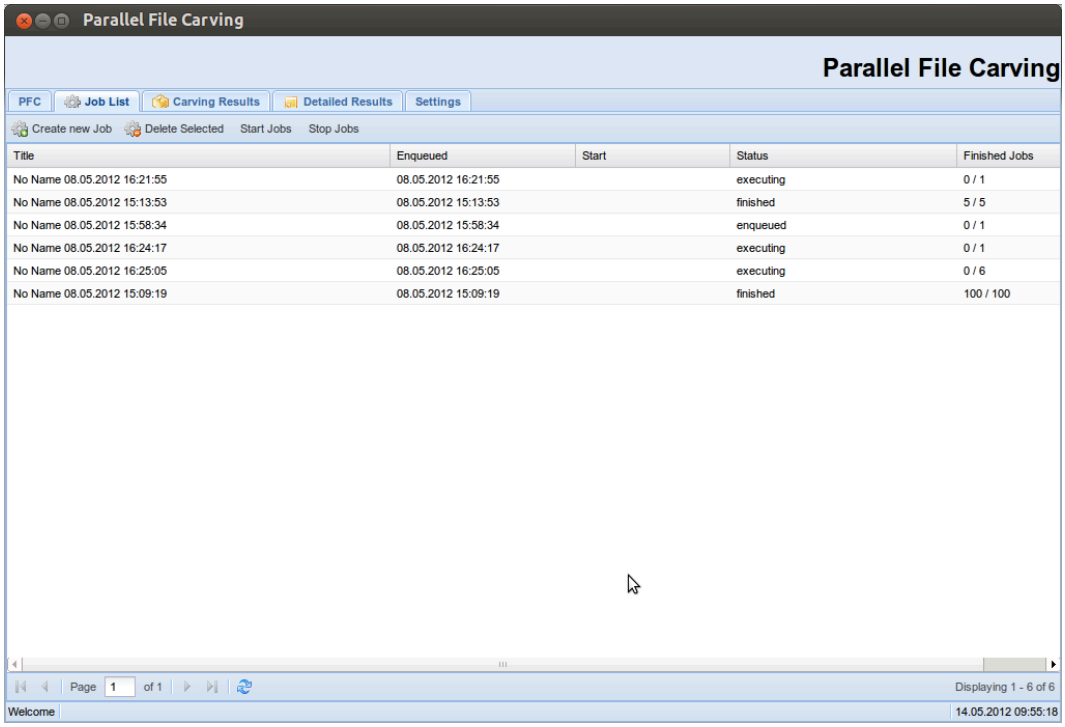
6.2.2 Starten einer Analyse

Anlegen eines Jobs

Mit einem Klick auf *Create new Job* unter dem Tab **Job List** (Abbildung 6.1(b)) wird ein neuer Job erzeugt. Dabei sind mehrere Schritte notwendig.

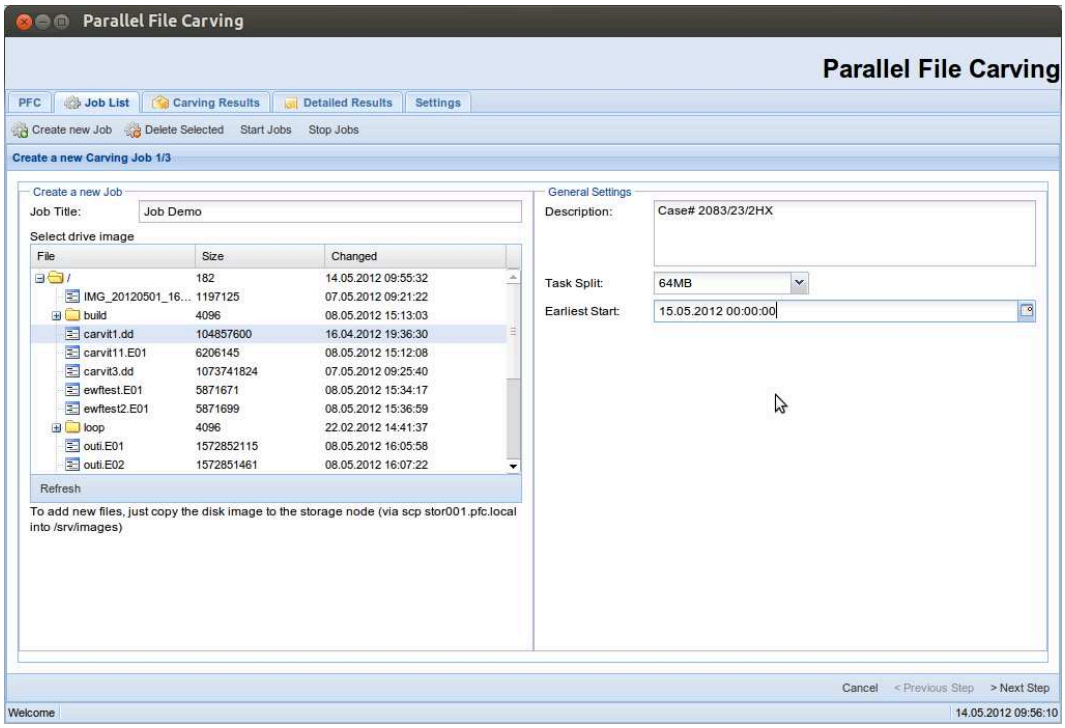


(a) Übersicht Parallelized File Carving

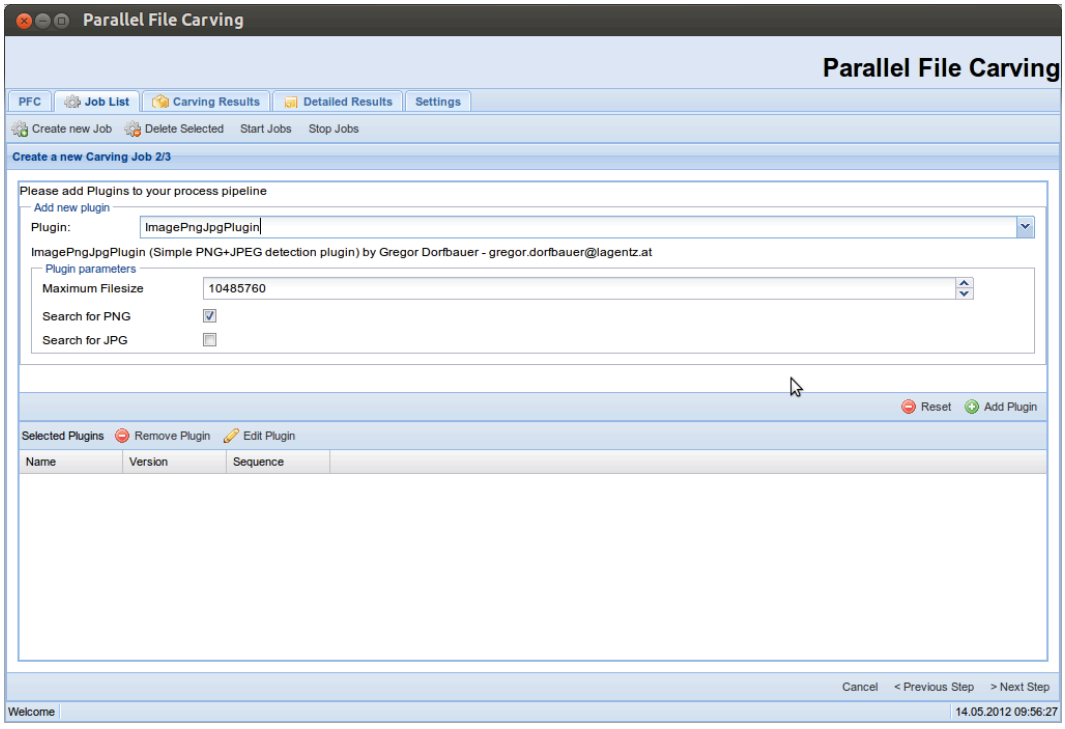


(b) **Job List**-Ansicht von PFC

Abbildung 6.1: Übersicht und Job-Liste



(a) Erstellen eines neuen Jobs, erster Schritt



(b) Erstellen eines neuen Jobs, Plugin-Einstellungen

Abbildung 6.2: Erstellen eines neuen Jobs mit Plugins

Allgemeine Einstellungen

Im ersten Schritt (Abbildung 6.2(a)) wird ein Titel vergeben und die Abbilddatei ausgewählt. In der rechten Spalte können eine Beschreibung, die Größe der Teilarbeitspakete und ein Datum zum frühestmöglichen Start vergeben werden. Die Bearbeitung des Jobs beginnt erst nach diesem Datum. Dadurch kann die Auslastung des Systems besser geplant werden. In dem Fall, dass kein Datum eingestellt wird, beginnt die Ausführung sofort.

Auswahl und Konfiguration der Plugins

Im nächsten Schritt (Abbildung 6.2(b)) können die Plugins, welche das ausgewählte Abbild analysieren sollen, ausgewählt werden.
Sobald ein Plugin ausgewählt ist, erscheinen die zu diesem Plugin benötigten Parameter und können eingestellt werden. Im unteren Bereich wird die Verarbeitungsreihenfolge der ausgewählten Plugins dargestellt.

Starten und Überwachen des Jobs

Ist zumindest ein Plugin ausgewählt, werden im letzten Schritt die Einstellungen noch einmal zusammengefasst. Ein Klick auf *Finish* legt den Job an und startet je nach Datumseinstellung die Bearbeitung sofort.

In der Liste unter **Job List** (Abbildung 6.1(b)) kann der Status des Jobs überwacht werden, in der Spalte *Finished Jobs* erhöht sich die Anzahl der bereits abgeschlossenen Arbeitspakete.

6.2.3 Auswertung

Nachdem ein Job abgeschlossen ist, können die Ergebnisse analysiert werden. Ein Doppelklick auf den Job unter **Job List** (Abbildung 6.1(b)) oder die Auswahl des Jobs in der Dropdown-Box unter **Carving Results** (Abbildung 6.3) lädt alle gefundenen Dateifragmente in die Ergebnisliste.

Dateivorschau

Ein Klick auf den Button *Show Preview* schaltet die automatische Bildvorschau (Abbildung 6.3) ein- und wieder aus. Im rechten Bereich öffnet sich ein Fenster, welche die aktuell ausgewählte Datei darstellt. Dieses Fenster wird automatisch aktualisiert,

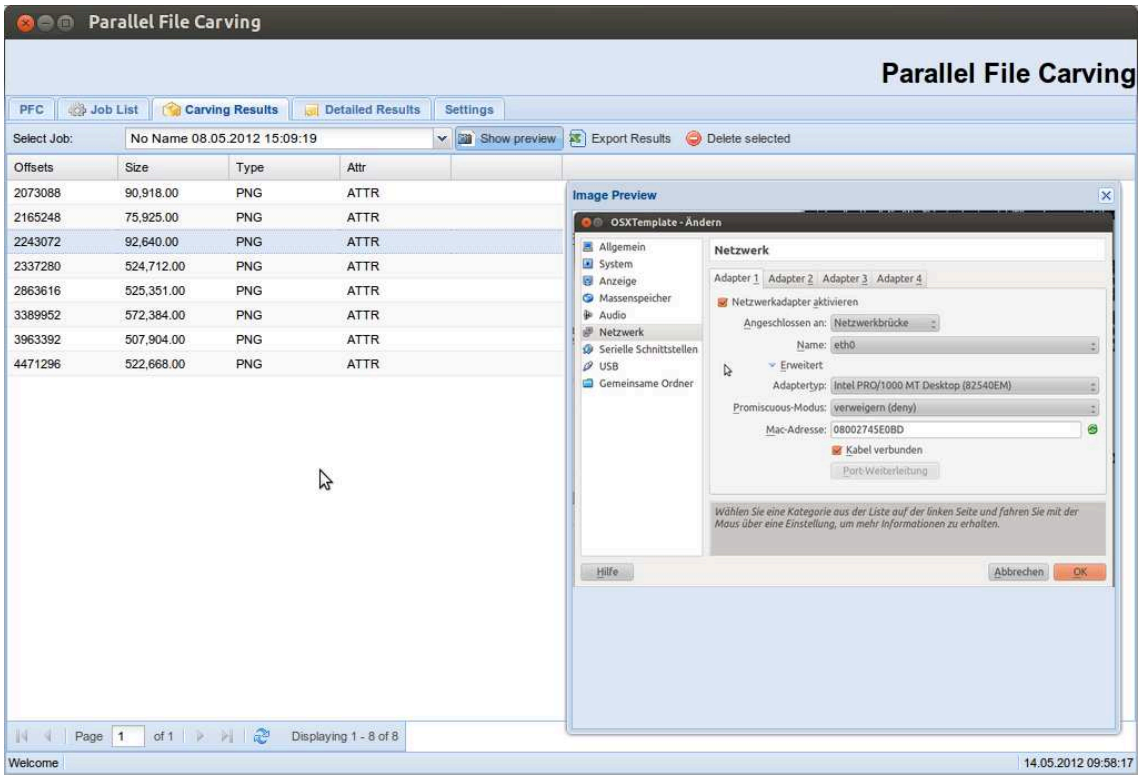


Abbildung 6.3: Ansicht der Ergebnisse mit automatischer Vorschau

wenn sich die Auswahl ändert. Der Anwender kann die Größe und Position des Fensters selbst bestimmen.
Dateien, welche nicht direkt im Browser angezeigt werden können, werden als Download-Link eingebunden, damit die Ergebnisse heruntergeladen werden können.

Export

Der Knopf *Export Results* erzeugt eine CSV-Datei mit den dargestellten Dateien und Fragmenten um eine weiterführende Analyse durch Skripts zu ermöglichen.

Die Struktur dieser CSV-Datei besteht aus folgenden Spalten:

- Das Feld **Job ID** stellt eine eindeutige Identifizierung des Jobs dar.
- In **Filename** wird der Dateiname des ursprünglichen Abbilds abgespeichert.
- In **Fragments** wird die Fragmentliste der Datei in einer zu CarvFS kompatiblen Syntax abgespeichert.

- Die Felder **Size** und **Type** bezeichnen die Größe und den Dateityp der gefundenen Datei.
- Das Feld **Attributes** ist für die Speicherung von Plugin-spezifischen Metadaten vorgesehen.

In Listing 6.4 sind exemplarisch zwei Einträge dieser Datei dargestellt.

```
1 Job ID,Filename,Fragments,Size,Type,Attributes
2 3f3e502b-e6e2-4f83-90f4-8a91e1c1bb8b,/fc/t_5g.dd,/CarvFS
    /10486784+203108.crv,203108,PNG,ATTR
3 3f3e502b-e6e2-4f83-90f4-8a91e1c1bb8b,/fc/t_5g.dd,/CarvFS
    /10691584+203108.crv,203108,PNG,ATTR
```

Listing 6.4: Inhalt einer Export-Datei

Mit *Export Files* können alle gefundenen Dateien auf einmal zur weiteren Verarbeitung exportiert werden. Dabei wird auf dem Server ein Verzeichnis angelegt und sämtliche Dateien werden als symbolischer Link mit Hilfe von CarvFS abgelegt.
Die Dateien können dann bequem per SSH/SCP auf den Analyserechner kopiert werden oder mit weiteren Werkzeugen direkt bearbeitet werden.

Detaillierte Ergebnisse auf Cluster-Ebene

Im Bereich **Detailed Results** können die gefundenen Dateien in einer Hex-Ansicht analysiert werden. Geübte Ermittler können so schnell feststellen, ob es sich um einen false-positive Fund handelt, wenn eine Datei nicht angezeigt werden kann.
Im Abschnitt 9.2 werden Erweiterungen zur Block-Bearbeitung bzw. Umsortierung der gefundenen Fragmente vorgestellt.

7 Entwicklerhandbuch

Der Funktionsumfang und die unterstützten Dateitypen und Dateisysteme können in Parallelized File Carving durch Plugins erweitert werden.

Sobald das Plugin vom System erkannt wurde, ist es im Webinterface möglich, das Plugin zu verwenden. Ein Plugin besteht aus einem eindeutigen Namen, einer Menge von Parametern zur Konfiguration des Plugins zur Laufzeit und zumindest einer Methode, welche einen gegebenen Datenblock untersucht.

Für die Plugins muss ein Wrapper-Skript in Python, wie nachfolgend erklärt, erstellt werden. Die eigentlichen Implementierungen der File-Carving-Algorithmen können dann in einer beliebigen Umgebung ausgeführt werden. Dazu wird das Plugin als dynamische Bibliothek erzeugt und von dem Wrapper-Skript aufgerufen.

7.1 Plugin-Schnittstelle

Die Plugin-Schnittstelle von Parallelized File Carving besteht aus drei Grundkomponenten. Allgemeine Informationen zum Plugin (Name, Beschreibung, Autor und Version) sollten angegeben werden, damit der Benutzer das Plugin leicht erkennen kann.

Die Definition der notwendigen Parameter für die Plugin-Abarbeitung (als Beispiel sei eine maximale Auflösung eines zu suchenden Bildes oder die Sprache eines Dokuments genannt) beschreibt die Möglichkeiten der Plugin-Konfiguration. Abschließend muss eine konkrete Verarbeitungsfunktion entwickelt werden, welche einen gegebenen Datenblock in einem Image untersucht und Dateien bzw. Dateifragmente findet.

Plugins für Parallelized File Carving müssen von der Klasse `PFCPlugin` abgeleitet werden.

7.1.1 Allgemeine Informationen zum Plugin

Um die allgemeinen Informationen eines Plugins zu setzen, müssen die Methoden `getVersion`, `getDescription` und `getAuthor` implementiert werden (siehe Listing 7.1).

```
1 class PluginName(PFCPlugin):
2   def getVersion():
3     return 1
4   def getDescription():
5     return "Beschreibung des Plugins"
6   def getAuthor():
7     return "Gregor Dorfbauer"
```

Listing 7.1: Allgemeine Informationen zu einem Plugin

Diese beschreibenden Informationen werden auch im Webinterface bei der Plugin-Übersicht angezeigt und helfen dem Benutzer, die Funktion des Plugins zu erkennen und eventuelle Einschränkungen auf Dateitypen sofort zu berücksichtigen.

7.1.2 Parameter

Die Verarbeitungsmethode in einem Plugin muss oft zur Laufzeit konfiguriert werden. Beispiele dafür sind die Sprache der zu suchenden Dokumente, eine Einschränkung auf eine gewisse Dateiversion oder ein Grenzwert der Entropie innerhalb eines Datenblocks.

Parametertypen

Parallelized File Carving unterstützt sechs grundlegende Parametertypen. Die definierten Parameter können jeweils mit einem Minimal- und Maximalwert, einem Standardwert bzw. einer Auswahlliste für mögliche Werte eingestellt werden.

Jeder Parameter erhält zu diesen Grenzwerten auch einen pro Plugin eindeutigen Namen und eine Beschreibung, welche im Webinterface unterstützend angezeigt wird.

- `PFCIntegerParam` - ein Parameter für Ganzzahlen. Dieser wird als Eingabefeld mit Inkrement- und Dekrement-Knöpfen dargestellt.
- `PFCFloatParam` - ein Fließkommaparameter, das Eingabefeld entspricht dem Integer-Parameter, unterstützt aber zusätzlich Kommazahlen als Eingabe.

- `PFCStringParam` - ein Zeichenkettenparameter zur Spezifikation von allgemeinen Parametern. Minimal- und Maximalwert entspricht hier der Länge der Zeichenkette.
- `PFCDateTimeParam` - ein Datumswert, welcher in einem Kalendereingabefeld dargestellt wird.
- `PFCBooleanParam` - ein boolscher Parameter, dargestellt als Checkbox.
- `PFCOneOfNParam` - ein Auswahlparameter, die angegebene Auswahlliste von möglichen Werten wird in einer Dropdown-Box dargestellt.

Definition von Parametern

Durch die Definition der Parameter werden auch automatisch Parameterdialoge im Webinterface erzeugt.
Das folgende, fiktive `ClearParameterTest`-Plugin in Listing 7.2 zeigt, wie alle Parametertypen im Plugin definiert werden. Die Anzeige der Parameterdialoge im Webinterface ist in Abbildung 7.1 gezeigt. Dieser Dialog wird angezeigt, sobald dieses Plugin im Webinterface ausgewählt wird.

```
1  class ClearParameterTestPlugin(PFCPlugin):
2    def __init__(self):
3      self.params = [
4      PFCIntegerParam("maxfilesize", "Maximum Filesize",
5      "Maximum search size for a single File", 10 * 1024 * 1024),
6      PFCFloatParam("entropylimit", "Entropy limit",
7      "Minimum entropy in a block", 0.6, 0.0, 1.0),
8      PFCStringParam("contentsearch", "Content search",
9      "Checks if content matches this string", "CONFIDENTIAL"),
10     PFCDateTimeParam("exifcreate", "EXIF create date after",
11     "Checks if EXIF date is after the specified date",
12     datetime.datetime.now()),
13     PFCOneOfNParam("searchalgo", "Search algorithm",
14     "Specifies the search algorithm to use",
15     ["FullRegex", "HeaderOnly", "FormatCheck"], "HeaderOnly"),
16     PFCBooleanParam("useheuristic", "Heuristic search",
17     "Use heuristic search", True)]
```

Listing 7.2: Definition von Parametern in einem Plugin

Abbildung 7.1: Automatische Darstellung der Plugin-Parameter

Zugriff auf Parameterwerte

Um zur Laufzeit auf Parameterwerte zugreifen zu können, gibt es die Methode `param`, ein Beispiel, welches den Parameter `maxfilesize` ausliest, befindet sich im Listing 7.3.

```
1 class ClearParameterTestPlugin(PFCPlugin):
2   def process(self, filepointer, offset, clustersize, end, fsize):
3     parameter = self.param("maxfilesize")
4     print "Maximum Filesize is %d" % (parameter)
```

Listing 7.3: Zugriff auf Parameterwerte zur Laufzeit

7.1.3 Verarbeitungsmethode in Plugins

Die Verarbeitungsmethode `process` erledigt die eigentliche Arbeit des Plugins.

In dieser Methode wird ein Bereich des Festplattenabbilds analysiert. Dazu wird die Methode für jeden Block genau einmal mit folgenden Parametern aufgerufen:

- `filepointer` - ein offener File-Pointer zu dem Festplattenabbild
- `offset` - ab welcher Position in dieser Datei soll begonnen werden
- `clustersize` - Cluster-Größe des verwendeten Dateisystems (falls bekannt)
- `end` - Ende des Suchbereichs
- `filesize` - Gesamte Dateigröße des Images

Sobald die Verarbeitungsmethode eine Datei, bestehend aus einem oder mehreren Fragmenten (dargestellt als Koordinaten innerhalb des Abbilds) gefunden hat (z.B. wie in

Abschnitt 2.7 auf Seite 16 für PNG Dateien gezeigt), kann ein Ergebnis abgespeichert werden.

Das Beispiel in Listing 7.4 speichert eine Datei vom Typ `PNG` mit zwei Fragmenten der Offsets `2000-5000` und `9000-9500`. Dazu wird eine Instanz der Klasse `PFCResult` angelegt und dieser Result-Klasse eine Datei mit einer Fragmentliste hinzugefügt.

```
1 class FragmentPlugin(PFCPlugin):
2   def process(self, filepointer, offset, clustersize, end, filesize)
      :
3     result = PFCResult()
4     parts = [{"start": 2000, "end": 5000}, {"start": 9000, "end":
        9500}]
5     result.addFile("PNG", parts)
6     return result
```

Listing 7.4: Abspeichern einer gefundenen Datei

7.2 Testen

Damit ein Plugin zeitsparend getestet werden kann, empfiehlt es sich, kleine Dateisystemabbilder selbst zu erzeugen. Unter Linux können leere Dateien, genauso wie Festplatten, formatiert werden. Solche Abbilder können anschließend eingehängt werden und mit Dateien gefüllt werden.

7.2.1 Anlegen eines künstlichen Dateisystemabbilds

Das Listing 7.5 zeigt, wie eine Abbilddatei erzeugt, formatiert und eingehängt wird. Anschließend werden einige Dateien in das Abbild kopiert und das Abbild wieder ausgehängt. Die so erzeugte Datei verhält sich wie ein Abbild einer Festplatte mit den darauf kopierten Dateien.

```
1 #Erzeugen einer leeren Datei mit 256MB
2 pfchost:~$ dd if=/dev/zero of=image.dd bs=1M count=256
3 #Erzeugen eines FAT-Dateisystems in der Datei
4 pfchost:~$ mkfs.vfat image.dd
5 mkfs.vfat 3.0.12 (29 Oct 2011)
6 #Einbinden des Dateisystems
7 pfchost:~$ mkdir loop
8 pfchost:~$ sudo mount -o loop image.dd loop
```

```
#Kopieren von Dateien in das Dateisystem
pfchost:~$ cd loop/
pfchost:~/loop$ sudo cp ../bild1.png ../bild2.png .
pfchost:~/loop$ ls
bild1.png bild2.png
pfchost:~/loop$ cd ..
pfchost:~$ sudo umount loop
```

Listing 7.5: Erzeugen eines künstlichen Dateisystemabbilds

Durch solche vorgefertigten Dateisystemabbilder muss während dem Entwicklungszyklus nicht auf vollständige Abbilder von mehreren 100 GB zurückgegriffen werden. Änderungen können somit schneller und einfacher getestet werden.

7.3 Tuning

In dem Fall, dass ein neu geschriebenes Plugin nicht schnell genug arbeitet, müssen einige Punkte vom Entwickler geprüft werden, bevor eine Optimierung vorgenommen wird.

7.3.1 Ein-/Ausgabe-Tuning

Für die Gesamtlaufzeit eines Plugins ist oft entscheidend, ob der Teil des Abbilds, welches das Plugin bearbeiten soll, mehrmals gelesen wird. Im Idealfall sollte das Plugin die Erkennung von Dateifragmenten in maximal zwei Durchläufen fertigstellen.

Das Lesen des Abbilds sollte wenn möglich in größeren Blöcken und linear erfolgen, d.h. ein Lesen des Abbilds Byte-für-Byte sorgt für längere Laufzeit als ein Lesen von Blöcken der Größe von einem Megabyte.

7.3.2 CPU-Tuning

Stellt sich die CPU als limitierender Faktor heraus, sollte der Plugin-Code zuerst mit einem Code Profiler (das Python-Modul `cProfile` ist hier geeignet) untersucht werden. Damit kann herausgefunden werden, ob bestimmte Teile des Plugins eventuell unnötig oft aufgerufen werden.

Ist die Optimierung des Plugins in Python nicht mehr zielführend, können zuvor durch

den Profiler entdeckte Hot Spots in einer C-Bibliothek adaptiert werden. Diese Bibliothek kann im Python-Plugin mit dem Python-Modul `ctypes` ohne zusätzliche Wrapper-Schicht verwendet werden.

8 Zusammenfassung

Die vorliegende Mastarbeit zeigt die Möglichkeiten der Parallelisierung von File-Carving-Prozessen. Diese Prozesse sind im Allgemeinen sehr zeit- und ressourcenintensiv. Die Parallelisierung führt zu kürzeren Analysezeiten und einer besseren Ausnutzung der bestehenden Rechenressourcen.

Nach einer ausführlichen Recherche des aktuellen Stands der Technik wurde zuerst eine fiktive Oberfläche entworfen, um die Anwendungsfälle simulieren zu können. Diese Analysephase lieferte wertvolle Ergebnisse, welche die Auswahl der Ziele und zu implementierenden Funktionen vereinfachten. Der erste Schritt zur konkreten Implementierung war ein Plugin, welches Bilddateien aus einem Dateisystemabbild extrahieren konnte. Durch diese Implementierung konnten die Anforderungen an die spätere Plugin-Schnittstelle gut aufgezeigt werden.

Nachdem dieses Plugin funktionsfähig war, wurde die Schnittstelle für die Plugins definiert und mit der Parallelisierung begonnen. Schon früh zeigte sich, dass die Erweiterung auf mehrere Hosts, welche als spätere Implementierung geplant war, durch die Architekturentscheidung bereits im Rahmen dieser Masterarbeit möglich wurde. Die Software wurde also von Grund auf für die Parallelisierung über mehrere Rechner geplant.

Die ersten erfolgreichen Tests der Parallelisierungsarchitektur zeigten eine Beschleunigung der Analyse im abgeschätzten Rahmen. An dieser Stelle wurde mit der Implementierung des Webinterfaces begonnen. Anpassungen an den geplanten Oberflächen haben noch zu kleineren Änderungen an der Plugin-Schnittstelle geführt, um auch weitere Parametertypen zu unterstützen und die automatische Dateivorschau zu ermöglichen.

Einige Funktionen zum Editieren von Ergebnissen und zusätzliche Plugins konnten nicht mehr im Rahmen dieser Masterarbeit bearbeitet werden und sind im folgenden Kapitel 9 gesammelt.

Im Zuge der schriftlichen Ausarbeitung dieser Masterarbeit wurde eine Zusammenfassung und Erklärung des Begriffs File-Carving erstellt. Dieser Abschnitt enthält sowohl eine grundlegende Definition (Kapitel 2) als auch eine Sammlung von fortgeschrittenen Carving-Techniken (Kapitel 3).

Damit Benutzer Parallelized File Carving einfach einsetzen können, wurden auch ein Benutzer- und Entwicklerhandbuch verfasst (Kapitel 6 und 7).

Diese Arbeit zeigt die Nützlichkeit der Parallelisierung und bietet darüber hinaus die Möglichkeit, Erweiterungen der Funktionalität einfach hinzuzufügen.

Abschließend soll auch die Steuerung der Software über ein plattformunabhängiges Webinterface hervorgehoben werden. Durch die JavaScript-Bibliothek ExtJS ist es möglich, Oberflächen direkt im Browser zu erzeugen, ohne dass der Benutzer ein Plugin installieren muss. Dadurch wird die Software universell einsetzbar und auch unabhängig gegenüber Betriebssystemwechseln und -aktualisierungen. Die Kommunikation des Browsers mit der Web-Anwendung erfolgt über AJAX-Aufrufe. Dadurch ist auch kein Neuladen der Seite notwendig, um Daten aktuell darzustellen.

9 Ausblick und mögliche Erweiterungen

Parallelized File Carving wurde als erweiterbares Framework entworfen. Dieser Abschnitt zeigt Möglichkeiten zur Erweiterung sowohl auf Plugin-Ebene als auch in Hinblick auf Erweiterungen des Kernsystems.

9.1 Plugin-Entwicklung

Wichtige Erweiterungsplugins betreffen die Unterstützung von Dateiformaten. Der Abschnitt 2.7 zeigt, wie anhand der PNG-Formatspezifikation Dateien gefunden werden können.

Die folgende, unvollständige Liste, zeigt Möglichkeiten zur Erweiterung von Parallelized File Carving durch neue Plugins.

9.1.1 Unterstützung von Office-Formaten

Neuere Versionen von Microsoft Office und LibreOffice sind im allgemeinen Fall XML-Strukturen, welche gemeinsam mit zusätzlichen Ressourcen als ZIP-Datei abgelegt sind.

Ein Plugin, welches diese Formate unterstützen soll, muss also ZIP-Dateien erkennen, den Inhalt der ZIP-Dateien analysieren und dadurch bestimmen, ob es sich um eine Office-Datei handelt. Eine Volltextsuche innerhalb der gefundenen Dateien ist ebenfalls denkbar.

9.1.2 Unterstützung von geschachtelten Dateistrukturen

Geschachtelte Dateistrukturen treten unter anderem bei gepackten Dateien (ZIP, RAR oder *.tar.gz-Archive) und Dateien innerhalb anderer Dateien auf. Eingebettete Dateien, als Beispiel seien Graphiken innerhalb eines Word-Textdokuments genannt, dürfen nicht als Ergebnisdateien geliefert werden. Im Idealfall werden diese Dateien in der korrekten Schachtelungsstruktur extrahiert.

Diese Erweiterung soll, sofern die Struktur des Behälters bekannt ist, auf andere Plugins zurückgreifen, um den Inhalt der eingebetteten Dateien zu erkennen.

9.1.3 Filterung von bereits bekannten Dateien

Diese Erweiterung setzt eine Datenbank von bereits bekannten Dateien voraus. Ansätze zur Erkennung von solchen Dateien sind im Abschnitt 3.4 dargestellt.

Bei der Implementierung dieses Plugins sind sowohl *known-bad-* als auch *known-good-*Dateien zu berücksichtigen. Durch die Filterung werden bekannte Dateien (Systemdateien oder Beispielbilder, welche auf vielen Systemen identisch vorkommen) von der späteren manuellen Analyse ausgeschlossen. Die Analyse kann somit schneller abgeschlossen werden.

Speziell im Carving-Prozess wäre auch eine Hashwert-Datenbank von Clustern oder Sektoren hilfreich, dadurch könnten direkt beim Carving-Vorgang existierende Blöcke ausgefiltert werden, ohne dass die Nachverarbeitung gestartet werden muss.

9.1.4 Analyse der Entropie von aufeinander folgenden Blöcken

Ändert sich die Entropie von aufeinander folgenden Blöcken abrupt, handelt es sich mit hoher Wahrscheinlichkeit um Blöcke, welche nicht zusammengehören (z.B. ein großes Textdokument gefolgt von einem Foto). Diese Information kann als zusätzliches, heuristisches Hilfsmittel zur Grenzenbestimmung verwendet werden.

Details zu diesem Ansatz befinden sich im Abschnitt 3.1.

9.1.5 Volltext-Suche

Oft ist man an Informationen innerhalb von Dateien interessiert. Diese Erweiterung bietet die Möglichkeit, bereits während dem Carving-Prozess die Abbilder nach Texten oder Mustern zu durchsuchen.

Das Plugin kann mit Hilfe von regulären Ausdrücken (Regular Expressions) jeden betrachteten Block durchsuchen und so Fragmente hervorheben, welche den gewünschten Suchtext beinhalten.

9.2 Editor-Schnittstelle für Blöcke

Nicht jede von einem Plugin gefundene Datei entspricht auch tatsächlich einer real existierenden Datei. Stimmt der Anfang oder das Ende von solchen False-Positive-Funden, dann kann die Datei durch eine Anpassung der Grenzen eventuell doch vollständig und korrekt wiederhergestellt werden.

Die Erweiterung umfasst eine graphische Schnittstelle im Webinterface, welche es dem Anwender ermöglicht, eine gefundene Datei auf Block-Ebene zu editieren. Zusätzlich zum Ändern von Dateigrenzen soll auch eine Zusammenfassung von zwei Dateien zu einer oder ein Aufteilen von einer Datei auf mehrere Teile möglich sein. Damit der Benutzer die Entscheidungen, welche Teile zusammengehören, treffen kann, müssen die Teile visualisiert werden können. Bei Klartextformaten (HTML, TXT, ...) ist sofort ersichtlich, ob Anfang und Ende zusammen gehören, bei Bilddateien ist eine Visualisierung der Teile anzustreben.

9.3 Integration von GlusterFS für Scale-Out

Die Anwendung von File Carving wird durch Parallelized File Carving stark beschleunigt. Theoretisch kann die Bearbeitung der Dateisystemabbilder mit jedem hinzugefügten Rechensystem schneller werden.

Flaschenhals bei der Geschwindigkeitssteigerung bleibt die Ein- und Ausgabegeschwindigkeit des Festplattensystems, welche nicht beliebig gesteigert werden kann. Ziel dieser Erweiterung wäre es, die bereits vorhandene Parallelisierung durch verschiedene Maschinen nicht nur auf CPU-Ebene zu nutzen, sondern auch eventuell vorhandenen lokalen Festspeicher zu verwenden.

9.3.1 GlusterFS

GlusterFS, ein Produkt der Firma Gluster (GlusterFS, 2012), welche Ende 2011 von Red Hat (RedHat, 2012) gekauft wurde, ist ein verteiltes Netzwerkdateisystem. Die Daten auf einem GlusterFS-Volume können entweder für gesteigerte Ausfallssicherheit redundant abgelegt werden, oder, für diesen Einsatzzweck interessanter, auf mehreren Rechnern aufgeteilt abgelegt werden. Dieser Betriebsmodus wird auch *Striped Storage* genannt.

Bei der Verteilung der Arbeitsaufgaben muss bei dieser Erweiterung Rücksicht genommen werden, damit die zu bearbeitenden Blöcke auch auf dem jeweiligen lokalen Speicher liegt. Durch diese Optimierung wird der Zugriff noch weiter beschleunigt, da die Daten nicht über das Netzwerk geholt werden müssen, sondern bereits direkt auf dem Host-Dateisystem vorliegen.
Die Zugriffsgeschwindigkeit auf ein GlusterFS-Volume steigt dadurch mit jedem zusätzlichen Speicherknoten an und die Analysezeit sinkt weiter.

Literaturverzeichnis

AcessData. *FTK der Firma AccessData: http://accessdata.com/, abgerufen am 25.11.2012.*

AMQP. 2012. *OASIS Advanced Message Queuing Protocol (AMQP) 1.0.* Online verfügbar unter: http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf, abgerufen am 25.11.2012.

AWS. 2012. *Amazon Web Services.* online erreichbar unter http://aws.amazon/com/ec2, abgerufen am 25.11.2012.

Baun, C. et al. 2010. *Cloud Computing - Webbasierte dynamische IT-Services.* Springer-Verlag Berlin Heidelberg.

Calhoun, W., & Coles, D. 2008. Predicting the types of file fragments. *In: Digital Investigation: The Proceedings of the eighth annual DFRWS conference, vol. 5; 2008; p14-20.*

CarvFS. 2010. *CarvPath library.* Online verfügbar unter: http://sourceforge.net/apps/mediawiki/carvpath/, abgerufen am 25.11.2012.

Celery. 2012. *Celery Distributed Task Queue.* Online verfügbar unter: http://celeryproject.org/, abgerufen am 25.11.2012.

Cohen, Michael I. 2007. Advanced carving techniques. *Digital Investigation 4*, September-December, 119–128.

Cohen, Michael I. 2008. Advanced JPEG carving. *Pages 1–6 of: e-Forensics '08: Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop.* ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Crockford, D. 2006. *The application/json Media Type for JavaScript Object Notation (JSON).* Internet Engineering Taskforce, Request for Comments: 4627, July 2006., http://www.ietf.org/rfc/rfc4627.txt, abgerufen am 25.11.2012.

Dean, J., & Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *In: OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation (Berkeley, CA, USA, 2004), USENIX Association, pp. 10–10.*

Duce, D. et al. 2003. *Portable Network Graphics (PNG) Specification (Second Edition).* Online verfügbar unter: http://www.w3.org/TR/PNG/, abgerufen am 25.11.2012.

ExtJS. 2012. *Sencha Ext JS library.* Online verfügbar unter: http://www.sencha.com, abgerufen am 25.11.2012.

Garfinkel, S. 2006. AFF: A New Format for Storing Hard Drive Images. *In: Communications of the ACM 49(2) (Feb. 2006), p85-87.*

Garfinkel, S. 2007. Carving contiguous and fragmented files with fast object validation. *In: Proceedings of the 2007 digital forensics research workshop, DFRWS, Pittsburgh, PA; August 2007.*

GlusterFS. 2012. Online verfügbar unter: http://www.gluster.org/, abgerufen am 25.11.2012.

GNU. 2012. *GNU coreutils.* online verfügbar unter http://www.gnu.org/software/coreutils/, abgerufen am 25.11.2012.

Harbour, Nicholas. 2006. *Department of Defense Computer Forensics Lab dd.* online verfügbar unter http://dcfldd.sourceforge.net/, abgerufen am 25.11.2012.

Karresand, M., & Shahmehri, N. 2008. Reassembly of Fragmented JPEG Images Containing Restart Markers. *In: Proc. of the 2008 European Conference on Computer Network Defense (EC2ND'08), Dublin, Ireland. IEEE, December 2008, pp. 25–32.*

Kolmogorov, A.N. 1965. Three approaches to the quantitative definition of information. *In: Problems Inform: Transmission 1 4-7, 1965.*

Koo, B. M. et al. 2012. A Study on Digital Forensic Software as a Service on Cloud Computing. *In: International Conference on Internet and Cloud Computing Technology (ICICCT) - Singapore.*

Kryder, Mark. 2005. Kryder's Law. Scientific American. Online verfügbar unter: http://www.scientificamerican.com/article.cfm?id=kryders-law, abgerufen am 25.11.2012.

Marziale, L. et. al. 2007. Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. *In: Digital Investigation, September 2007.*

Memon, N., & Pal, A. 2006. Automated reassembly of file fragmented images using greedy algorithms. *In: IEEE Transactions on Image Processing, 2006.*

Metz, J. 2012. *Expert Witness Compression Format specification.* Online verfügbar unter: http://libewf.googlecode.com/files/Expert0(EWF).pdf, abgerufen am 25.11.2012.

Metz, J. et al. *libewf.* Online verfügbar unter: http://sourceforge.net/projects/libewf/, abgerufen am 25.11.2012.

MongoDB. 2012. *Mongo DB, Document Database.* Online verfügbar unter: http://www.mongodb.org/, abgerufen am 25.11.2012.

Moore, G.E. 1965. Cramming more components onto integrated circuits. Electronics Magazine.

NIST, National Institute of Standards, & Technology. 2012. National Software Reference Library. Verfügbar unter http://www.nsrl.nist.gov/Downloads.htm, abgerufen am 25.11.2012.

Pylons. 2010. *Pylons web framework.* Online verfügbar unter: http://www.pylonsproject.org/projects/pylons-framework/about, abgerufen am 25.11.2012.

pymongo. 2012. *Python MongoDB Database driver.* Online verfügbar unter: http://pypi.python.org/pypi/pymongo/, abgerufen am 25.11.2012.

RabbitMQ. 2012. *RabbitMQ - Messaging that just works.* Online verfügbar unter: http://www.rabbitmq.com/, abgerufen am 25.11.2012.

RedHat. 2012. Online verfügbar unter: http://www.redhat.com/, abgerufen am 25.11.2012.

Roussev, V., & Richard III, G.G. 2004. Breaking the performance wall: The case for distributed digital forensics. *In: Proceedings of the 2004 Digital Forensics Research Workshop.*

Scharinger, J. 2012. *Vorlesungsunterlagen aus Kryptographie im Sommersemester 2012.*

Sonntag, M. 2009. *Disk/File system investigation.* Unterlagen zu IT-Recht und Computerforensik, online verfügbar unter http://www.fim.uni-linz.ac.at/lva/IT_Recht_Computerforensik/ws2009/Filesystem%20investigation.pdf, abgerufen am 25.11.2012.

Strozzi, C. 1998. *NoSQL: a non-SQL RDBMS.* Online verfügbar unter: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, abgerufen am 25.11.2012.

Veenman, J. 2007. Statistical disk cluster classification for file carving. *In: IEEE third international symposium on information assurance and security.*

Vrubel, A. 2011. Creation and Maintenance of MD5 Hash Libraries, and their Application in Cases of Child Pornography. *In: The sixth international conference on forensic computer science, 2011.*

Lebenslauf

Name: Gregor Dorfbauer, BSc
E-Mail: gregor.dorfbauer@aon.at

Schulbildung
1993 - 1997 Volksschule Perg
1997 - 2001 Hauptschule I Perg
2001 - 2006 Höhere technische Bundeslehranstalt für Elektronik (Ausbildungsschwerpunkt: Technische Informatik) Steyr - mit Auszeichnung abgeschlossen

Zivildienst
2006 - Allgemeines Krankenhaus der Stadt Linz

Studium
Oktober 2007 - September 2010 Bachelorstudium Informatik (JKU Linz) - mit Auszeichnung abgeschlossen
Seit September 2010 Masterstudium Informatik (JKU Linz)

Berufserfahrung
Seit 2007 selbstständig mit La Gentz KG

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Schwertberg, Februar 2013

Gregor Dorfbauer