Einführung in die Analyse von Embedded-Firmware

Tim Krüger, Hochschule Niederrhein

28.04.2025

Zur Person

Tim Krüger

- Masterstudent der Informatik
- Mitarbeiter am Fachbereich 03 (Informatik und Elektrotechnik)
  - Praktische Informatik unter Prof. Gref
  - Zuletzt im Projekt *Übungsautomatisierung*
- Vorher 3 Jahre in der Embedded-Industrie gearbeitet
  - Deeply-Embedded, STM32-Mikrocontroller
- Masterarbeit bei Prof. Quade
  - Digitalforenische Analyse von Embedded-Firmware

Inhaltsverzeichnis

Inhaltsverzeichnis

Grundlegende Begriffsklärung

Firmware

Firmware ist eine spezielle Art von Software, die direkt auf der Hardware eines eingebetteten Systems ausgeführt wird. Sie ist für die Initialisierung, Steuerung und Funktionalität der Hardware verantwortlich. Im Gegensatz zu regulärer Software ist Firmware eng an die jeweilige Hardware gebunden und wird in nichtflüchtigem Speicher wie Flash abgelegt.

Beispiele:

- Firmware eines Routers
- Firmware in IoT-Geräten wie z.B. Thermostaten oder Türschlössern
- BIOS eines PCs

Open- vs: Deeply-Embedded Firmware

Open-Embedded Firmware

- Basieren auf Linux
- Klare Trennung von Kernel und Userland
- Verwendung zusätzlicher Dateisysteme (z.B. SquashFS oder ext4)

Deeply-Embedded Firmware

- Kein OS oder nur leichtgewichtiges RTOS
- Monolithischer Binärblob
- Kein klassisches Dateisystem

Image

Ein Image ist ein vollständiges Abbild eines Speicherbereichs oder Datenträgers, das die exakte Byte-für-Byte-Repräsentation enthält. Im Embedded-Bereich kann es sich z.B. um ein Flash-Abbild handeln, das Partitionstabellen, Bootloader, Kernel und Root-Dateisystem enthält.

Ein binäres Abbild (Image) kann mittels `dd` von bspw. einer Speicherkarte extrahiert werden:

```
dd if=/dev/sdX of=firmware.img bs=4M status=progress
```

Partitionen

Partitionen sind logisch getrennte Bereiche innerhalb eines Speichermediums. Sie strukturieren den Speicher in unabhängige Segmente. Jede Partition erfüllt eine spezifische Aufgabe.

Typische Partitionen in Embedded-Firmware:

- boot
- kernel
- rootfs
- data
- nvs
- factory

Partitionstabellen

Die Partitionstabelle beschreibt Lage, Typ und Größe der einzelnen Partitionen. Je nach Plattform existieren unterschiedliche Formate:

- MBR (Master Boot Record)
  - 512 Byte am Anfang des Mediums
  - Für ältere oder einfache Systeme
- GPT (GUID Partition Table)
  - Moderner Standard für große Images und UEFI-Systeme
- Custom Formate
  - Bspw. nutzt der ESP32 eine individualisierte Partitionstabelle

Dateisystemformate

Dateisystemformate beschreiben, wie Dateien innerhalb einer Partition organisiert sind. Es existieren viele verschiedene Formate:

Format	Beschreibung
ext2/3/4	Klassische Linux-Dateisysteme
FAT12/16/32	Einfaches Dateisystem, weit verbreitet
SquashFS	Read-only, hochkomprimiert
JFFS2	Flash-freundlich, log-basiert
UBIFS	Nachfolger von JFFS2, für größere NAND-Flash
CramFS	Altes, kleines read-only FS
exFAT	Microsoft FAT-Nachfolger

Mounting

Mounting beschreibt den Vorgang, ein Dateisystem innerhalb eines Images oder einer Partition in das laufende Betriebssystem einzuhängen. Der Inhalt kann dann wie ein normales Verzeichnis durchsucht werden.

Beispiel:

```
sudo mount -o loop firmware.img /mnt/rootfs
```

Entropie (1)

Entropie ist ein Maß für die Zufälligkeit von Daten in einer Datei. Sie gibt Hinweise auf Komprimierung oder Verschlüsselung. Eine hohe Entropie (nahe 8) spricht für verschlüsselte oder stark komprimierte Inhalte.

- Gemessen wird mittels der Shannon-Entropie
- Binwalk kann diese grafisch darstellen (normiert auf 1)

Beispiel:

```
binwalk -E firmware.img
```

Entropie (2)

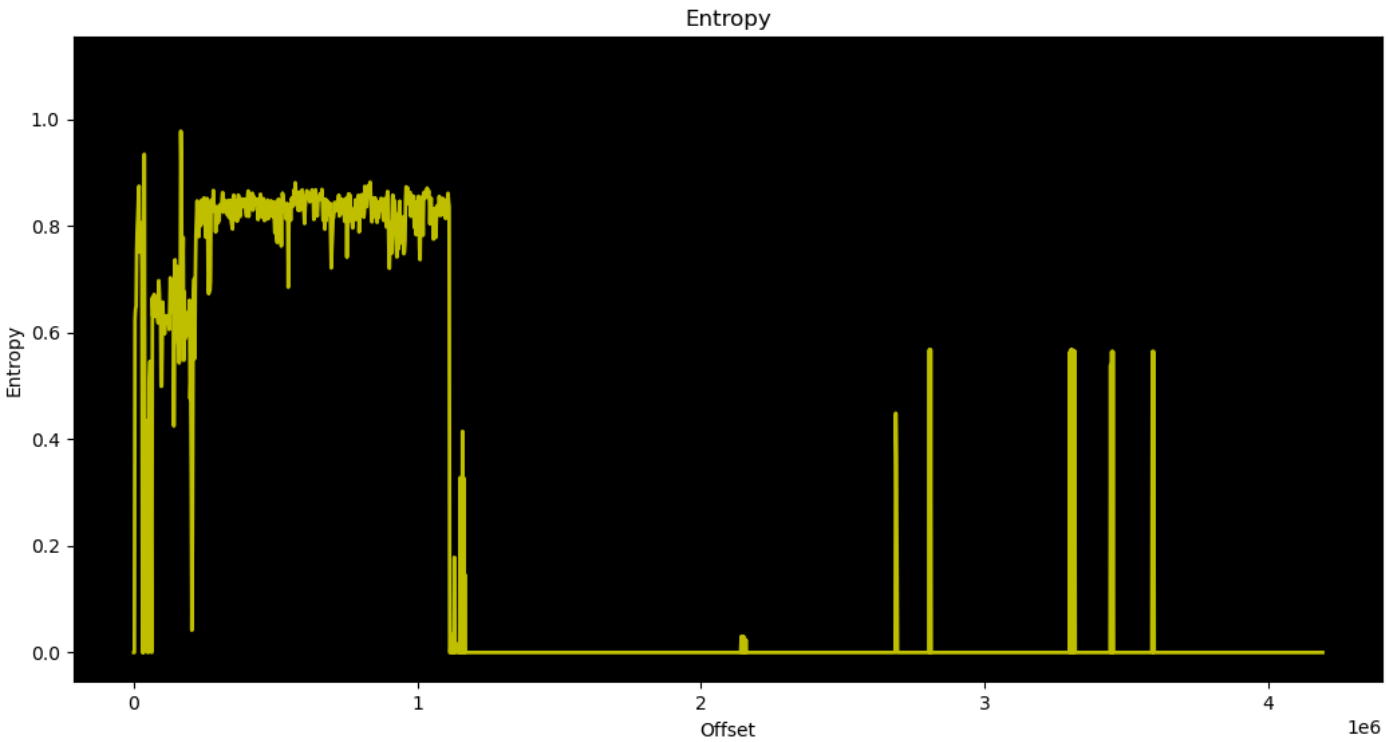

Figure 1: Entropie ESP32-Image

- Interessanter Bereich von 0 bis ungefähr 1.1 MB

QEMU

QEMU (Quick Emulator) ist ein Open-Source-Emulator zur Ausführung von Software für andere Architekturen. Es existieren zwei Betriebsmodi:

- `qemu-system-arch`: Komplette Emulation eines Systems (z.B. ARM-basiert)
- `qemu-user-static`: Ausführung einzelner Binärdateien innerhalb einer chroot-Umgebung

Statische vs: Dynamische Analyse

Statische Analyse

Untersuchung der Firmware ohne Ausführung.

- Fokus auf Dateien und Formate
- Gezielt, schnell, einfach reproduzierbar

Dynamische Analyse

Untersuchung der Firmware während Ausführung.

- Emulation oder nativ
- Laufzeitverhalten, System- und Bibliotheksaufrufe, Netzwerkaktivität, Self-Modifying Code ...

Datenextraktion

Aufbau der Analyse

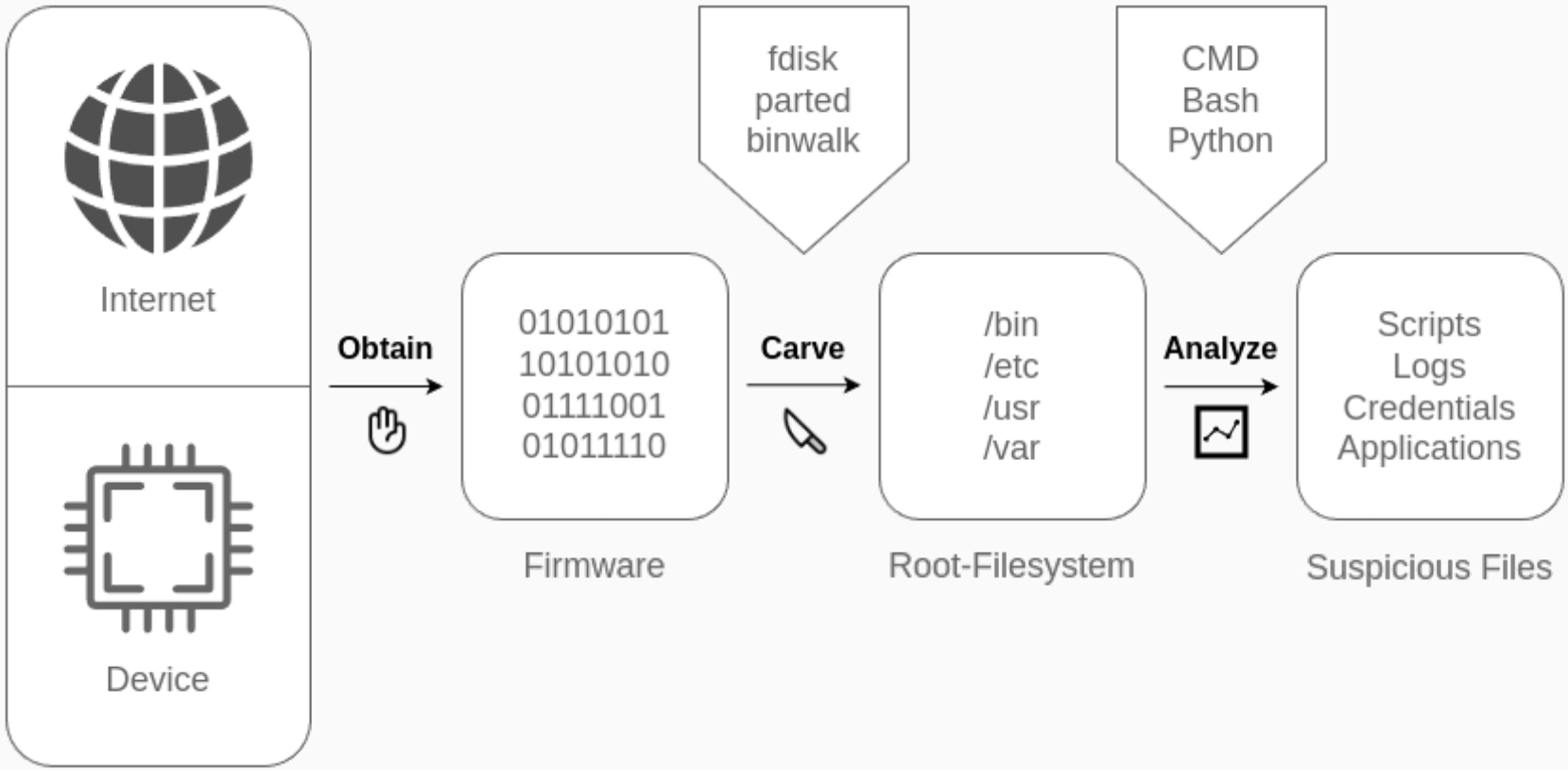

Figure 2: Vorgehensweise

Vorbereitung (1)

Hardware

Welche Hardware liegt konkret vor?

- Erste Einschätzung der Applikation auf Basis von Leistungsfähigkeit

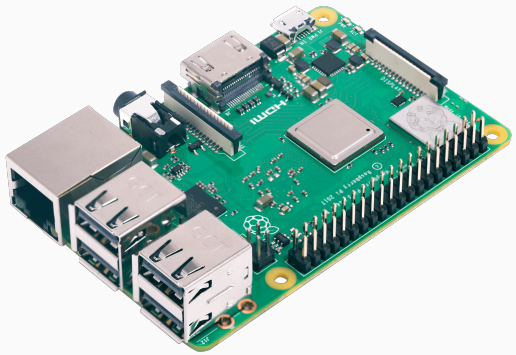

Figure 3: Raspberry Pi

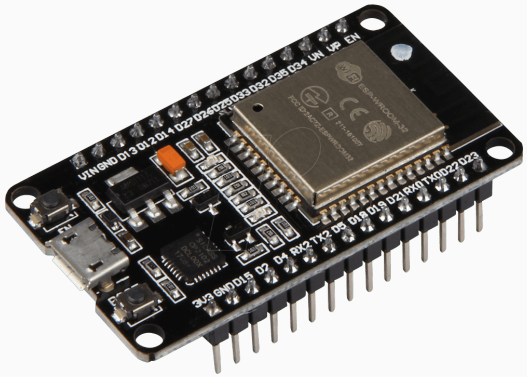

Figure 4: ESP32

Vorbereitung (2)

Firmware

Wie liest man die Firmware aus?

- Raspberry Pi Speicherkarte
- ESP32-Tools
- Herstellerwebsite konsultieren
- Freizugängliche Firmware (für z.B. Router-Updates)

Deeply-Embedded Firmware ist komplizierter.

- Debug-Interfaces (ggf. externe Hardware-Debugger)
- Flash entlöten und auslesen
- Sidechannel-Angriffe

Extraktion des Binary-Blobs

Raspberry Pi

- Speicherkartenpartition (z.B. `/dev/sdb1`) mounten
- Kopie auf Byte-Ebene erstellen
- Vorsicht bei nativer Ausführung unter Linux

```
dd if=/dev/sdX of=firmware.img bs=4M status=progress conv=fsync
```

ESP32

- ESP32-Tools verwenden
- Einige Pins müssen ggf. auf GND gezogen werden
- Klappt nur wenn Flash-Encryption ausgeschaltet ist

```
esptool.py -p PORT -b BAUD_RATE read_flash START END firmware.img
```

Sichern einer forensischen Kopie

- Bitgenaue Kopie der Firmware erstellen (vorherige Folie)
- Hashwert berechnen und abspeichern

```
sha256sum firmware_master.img > firmware_master.img.sha256
```

- Masterdatei auf Read-Only setzen

```
chmod 444 firmware_master.img
```

- Arbeitskopie erstellen

```
cp firmware_master.img firmware_working.img
```

Werkzeuge für diesen Arbeitsschritt

Tool	Beschreibung
`ls`	Anzeige von Dateien
`cat`	Anzeige von Inhalten
`file`	Typ-Identifikation über Magic Bytes
`mount`	Einhängen von Partitionen
`chmod`	Setzen von Datei-/Verzeichnisrechten
`cp`	Kopieren von Dateien oder Verzeichnissen
`dd`	Byte-genaues Kopieren/Carving
`esptool.py`	Kommunikation mit ESP32-Bootloader
`sha256sum`	Berechnung einer SHA-256-Hashsumme

Interpretation des Binary-Blobs

Klassifikation

Open-Embedded

- Große Dateien
  - MB, u.U. etliche GB
- Klare Partitionierung
- Dateisysteme
- Oft höhere Entropie
  - Komprimierung
  - Verschlüsselung
- Architekturen
  - *ARM Cortex-A*
  - *x86*

Deeply-Embedded

- Sehr geringe Dateigröße
  - KB, max. wenige MB
- Keine klaren Partitionen
  - Binärer Monolith
- Keine Dateisysteme
- Oft niedrige Entropie
  - Bootloader
  - Hardware-Register
  - Memory-Mapped-IO
- Architekturen
  - *ARM Cortex-M*
  - *STM32*

Signaturen

Erste Hinweise kann man mit der Auswertung von Strings und Signaturen erlangen.

Open-Embedded

- Linux-spezifische Pfade (z.B. */bin/sh*)
- Linux-Tools und Programme

Deeply-Embedded

- Hinweise auf Compiler, Linker, Echtzeitbetriebssystem
  - *ARM-GCC*, *FreeRTOS*, ...

```
strings firmware_working.img
```

Partitionstabelle

Die Partitionstabelle gibt Aufschluss darüber:

- Wie viele Partitionen es gibt
- Wo sich diese im Image befinden (Offset)
- Wie groß die einzelnen Partitionen sind
- Welchen Typ (bspw. Dateisystem) sie haben

Lokalisierung und Auswertung der Partitionstabelle hat deswegen oberste Priorität.

Analyse der Partitionstabelle

- Erste Analyse mittels `file`
  - Erkennt oft MBR/GPT Partitionen und Startsektoren
- MBR/GPT: Weitere Analysen mittels `fdisk` und `parted`
  - Auflisten der Partitionen mit Startsektor, Typ und Größe

Das klappt aber nicht immer ...

- Generische Analyse mittels `binwalk`
  - Erkennt Partitionen, Dateisysteme, und Header
  - ggf. auch automatisierte Extraktion möglich (Parameter `-e`)
- Einige Embedded-Systems haben Custom-Partitionstabellen
  - Oft bei fixen Offsets im Speicher
    - Bspw. der ESP32
  - Konsultieren der Herstellerwebsite

Zerlegung in Abschnitte (Carving)

Wenn interessante Partitionen (bspw. ein Root-Dateisystem) lokalisiert wurden, können diese gezielt extrahiert werden.

- Man nennt diesen Prozess *Carving* (Herausschneiden)
- Dabei *schneidet* man auf Byte-Ebene entsprechend von Größe und Offset die gewünschte Partition aus dem Image heraus
- **Wichtig:** Auf gegebene Sektorgrößen achten

Beispiel

```
Device       Boot   Start      End  Sectors  Size Id Type
backup.img1  *       2048  1050623  1048576  512M  c FAT32
backup.img2      1050624 15431646 14381023  6.9G 83 Linux

dd if=backup.img of=rootfs.img skip=1050624 bs=512 status=progress
```

Werkzeuge für diesen Arbeitsschritt

Tool	Beschreibung
`strings`	Extrahiert druckbare ASCII-/Unicode-Zeichen
`file`	Typ-Identifikation über Magic Bytes
`fdisk`	Listet Partitionstabellen (MBR/GPT) auf
`parted`	Analysiert Partitionstabellen (MBR/GPT)
`binwalk -E`	Entropieanalyse (graphischer Output)
`binwalk`	Sucht eingebettete Datenstrukturen
`binwalk -e`	Extrahiert automatisch eingebettete Datenstrukturen
`dd`	Byte-genaues Kopieren/Carving

Interpretation der Partitionen

Partitionen

Man findet verschiedenste Partitionen in einem Embedded-Image. Typisch sind zum Beispiel:

- boot
- kernel
- rootfs
- data
- nvs
- factory

Wenn man nicht weiß, wofür eine Partition benötigt wird $\rightarrow$ Google, Herstellerwebsite oder ChatGPT konsultieren.

Mounting eines Root-Dateisystems

Das Root-Dateisystem ist i.d.R. am interessantesten.

Inspektion

- Nach der Extraktion kann das Root-Dateisystem mit `file` und `debugfs` inspiziert werden
- Falls Reparaturen notwendig sind, steht einem u.A. das Werkzeug `fsck.ext4` zur Verfügung

Mounting

- Final kann das Root-Dateisystem gemountet werden

```
mkdir /mnt/rootfs
sudo mount -o loop rootfs.img /mnt/rootfs
```

Linux: Das Root-Dateisystem (1)

Bevor wir mit der Interpretation der Partitionen weitermachen, folgt ein kurzer Einschub zum Linux-Dateisystem. Wichtig für:

- Navigation und Zurechtfindung im gemounteten Dateisystem
- Trennung zwischen System- und Nutzerdaten
- Identifikation relevanter Datenquellen

Ein Linux-Dateisystem folgt dem sogenannten *Filesystem Hierarchy Standard (FHS)*. Dieser definiert, welche Verzeichnisse es gibt und wofür diese gedacht sind.

Es folgt keine vollständige Aufstellung. Eine gute Übersicht bietet die *Linux Foundation* beispielsweise hier.

Linux: Das Root-Dateisystem (2)

Verzeichnis	Bedeutung
`/bin`	Essenzielle Systemprogramme
`/sbin`	Systemadministrationsprogramme
`/etc`	Konfigurationsdateien
`/lib`	Bibliotheken für Programme (für `/bin` und `/sbin`
`/usr`	Zweite Hierarchie für nicht essentielle Anwendungen und Tools
`/home`	Nutzerverzeichnisse (optional)
`/var`	Variable Daten wie Logs, Spools, temporäre Daten

Spools sind Wartenschlangenverwaltungen für Dienste (bspw. *Cups*).

Linux: Das Root-Dateisystem (3)

Einige dieser Verzeichnisse sind in Embedded-Systemen abgespeckt oder fehlen komplett.

- `/usr` kann leer sein
- `/var` kann deutlich reduziert sein
- ...

Zusätzlich typisch:

- BusyBox als Ersatz für viele klassische Binaries (einzelne kompakte Binary)
- Minimale `init`-Skripte anstelle von `systemd`

Linux: Autostart

SysVinit

- Historisches Init-System, sequentiell und einfach
- Geringer Ressourcenbedarf
- Konfiguration über Skripte in `/etc/init.d/`
  - `/sbin/init` führt diese dann sequentiell aus

systemd

- Standard in vielen modernen Desktop- und Open-Embedded-Distributionen
- Parallele Initialisierung, Abhängigkeitsauflösung
- Höherer Ressourcenbedarf
- Konfiguration via `/etc/systemd/system/*.service`
- Aktivierung via `sudo systemctl enable *.service`

Analyse eines Root-Dateisystems (1)

Bei der Analyse des Dateisystems sind der Kreativität keine Grenzen gesetzt. Ein paar Anregungen:

- Analyse von Benutzerdaten
  - Passworthashes
  - Hardcoded Credentials in Skripten oder Configs
- Stringbasierte Suche von IPs und URIs (`grep`, `find`)
- Analyse von Init-Skripten
- Analyse von Services
  - Vor allem Netzwerkdienste
  - Unverschlüsselte Protokolle, offene Ports
- Analyse von SSH- und Telnet-Zugängen
- Auswertung von Logging-Dateien

Analyse eines Root-Dateisystems (2)

Folgende Inhalte sind oft besonders interessant:

- Inhalte von `/etc`, `/init`, `/var`, ...
  - `/etc/passwd`
  - `/etc/shadow`
  - `/etc/init.d`
  - ...
- Dienste: `inetd`, `telnetd`, `dropbear`, `sshd`, ...

Zur Auswertung stehen einem die bekannten Linux-Tools, Bash-Scripting oder auch Python-Scripting zur Verfügung.

Analyse eines Root-Dateisystems (3)

Auch die Shell kann einem als Informationsquelle dienen.

~/.profile

- Wird automatisch bei Login-Shells ausgeführt
- Enthält benutzerspezifische Umgebungsvariablen, Aliase, Pfadänderungen etc.
- Hinweise auf manuelle Modifikation, Nutzung, Debugging, ...

~/.history

- Enthält die letzten ausgeführten Shell-Kommandos
- Extrem nützlich zur Rekonstruktion von Benutzeraktionen
- Abhängig vom Shell-Typ (`bash`, `zsh`, ...)

Automatisierte Auswertung mittels Python

```
PASSWORD_FILES = {
    'unix_passwd': 'etc/passwd',
    'unix_shadow': 'etc/shadow',
}

def find_and_parse_password_files(rootfs_path):
    credentials = {}

    for key, relative_path in PASSWORD_FILES.items():
        full_path = os.path.join(rootfs_path, relative_path)

        if os.path.isfile(full_path):
            // Parse users
            // Parse password hashes

    return credentials
```

Automatisierte Auswertung mittels Bash

Einen guten Startpunkt bietet hierbei das Projekt firmwalker.

- Simples Bash-Skript
- Open-Source
- Output ist eine Textdatei

Durchsucht ein gemountetes Filesystem nach interessanten Dateien.

- Konfigurationen
- Skripte
- Credentials
- URLs und IPs
- ...

Werkzeuge für diesen Arbeitsschritt

Tool	Beschreibung
`file`	Typ-Identifikation über Magic Bytes
`debugfs`	Interaktives Dateisystem-Debugging für ext2/3/4
`fsck.ext4`	Konsistenzprüfung und Reparatur eines ext4-Dateisystems
`mount`	Hängt Partitionen/Images in das Dateisystem ein
`grep`	Durchsucht Dateien oder Ausgaben nach Strings
`find`	Durchsucht Verzeichnisse nach Dateien
`stat`	Zeigt Metadaten einer Datei (Zeitstempel, Rechte)
`ls -l`	Listet Dateien mit Rechten und Besitzern auf

Analyse- und Forensik-Frameworks

Übersicht

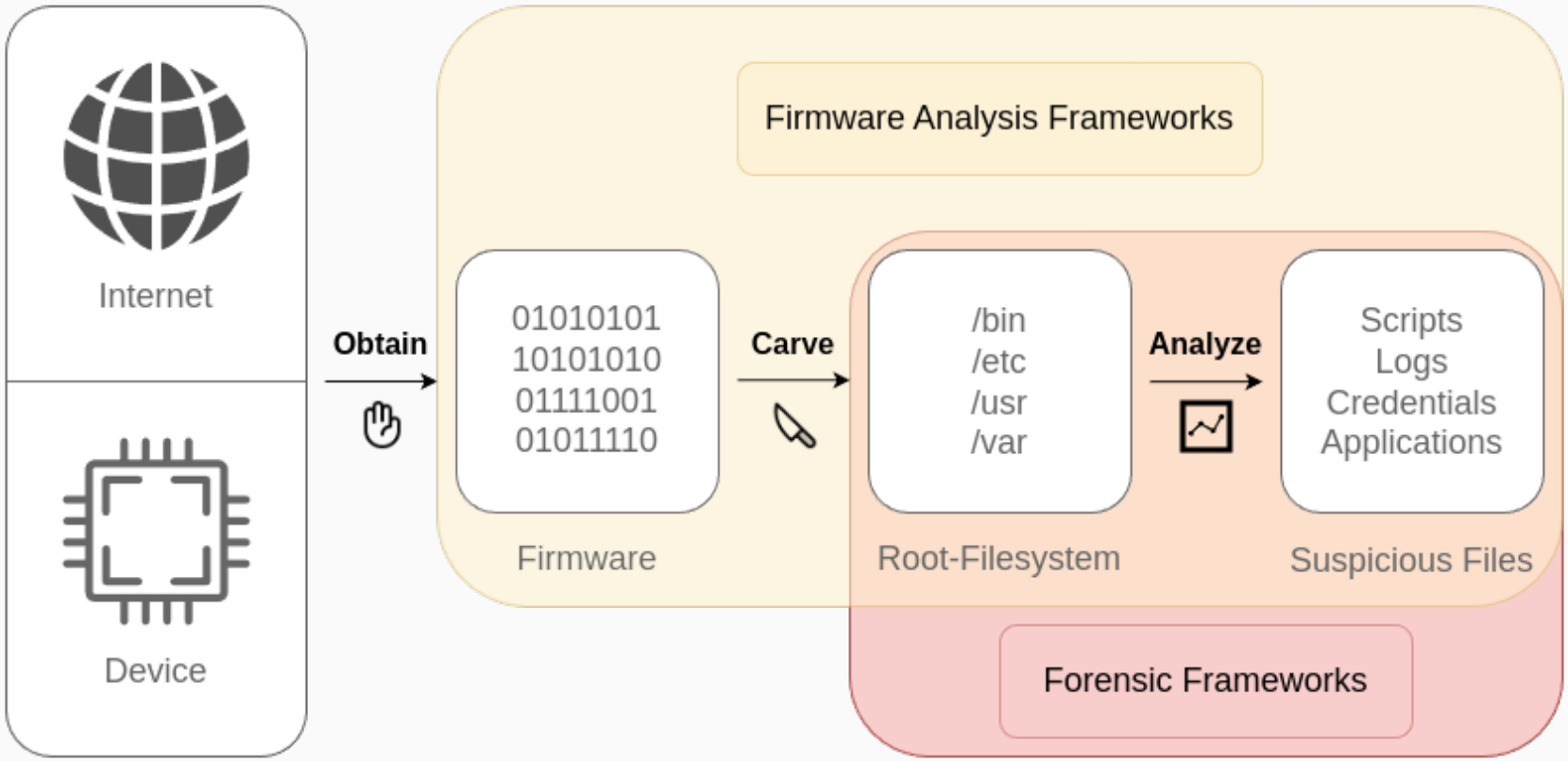

Figure 5: Analyse-Frameworks

Hintergrund

Für eine automatisierte Auswertung von Firmware-Images stehen einem Analyse-Frameworks zur Verfügung. Nach meinem bisherigen Kenntnisstand lassen sich diese grob in zwei Kategorien einordnen (wobei es Überschneidungen gibt).

- Firmware-Analyse Frameworks
- Forensik-Frameworks

Beide Arten von Frameworks stellen zusätzliche Funktionalitäten bereit, welche weit über die in Abbildung 4 vorgestellten Analyseschritte hinausgehen und dementsprechend den Rahmen dieser Präsentation sprengen.

Firmware-Analyse Frameworks

- Softwarebibliotheken für eine allgemeine Auswertung von (Embedded-)Firmware
- Durchsuchung und Analyse gefundener Root-Dateisysteme ist oft rudimentärer
  - Eher auf das Finden von Sicherheitslücken ausgerichtet
- Enthalten Komponenten zur Automatisierung des *Carving*-Prozesses
  - Softwarewrapper um `binwalk`
- Teilweise auch Emulationsfähigkeiten

Beispiel

- FACT

Forensik-Frameworks

- Unterstützen bei der Auswertung von Standard-Datenträgern und Root-Dateisystemen
- Automatisieren die Suche und Analyse verdächtiger Dateien
  - Oft unter speziellen, digitalforensischen Gesichtspunkten
- Meist nicht für die Extraktion von (komprimierten) Firmware-Images geeignet
  - Manuelle Extraktion vorher notwendig

Beispiel

- Autopsy

FACT (1)

FACT (Firmware Analysis and Comparison Tool) ist ein, am Fraunhofer FKIE entwickeltes, modulares Open-Source-Framework für die automatisierte Firmwareanalyse.

- Extraktion von Images mittels `binwalk`-Integration
- Vergleich von Firmware-Versionen (Delta-Analyse)
- Auswertung von Symboltabellen, Init-Skripten, Diensten ...

Architektur:

- Webfrontend für Nutzerinteraktion
- Backend in Python
- Datenbank zur Speicherung aller Analysen

FACT (2)

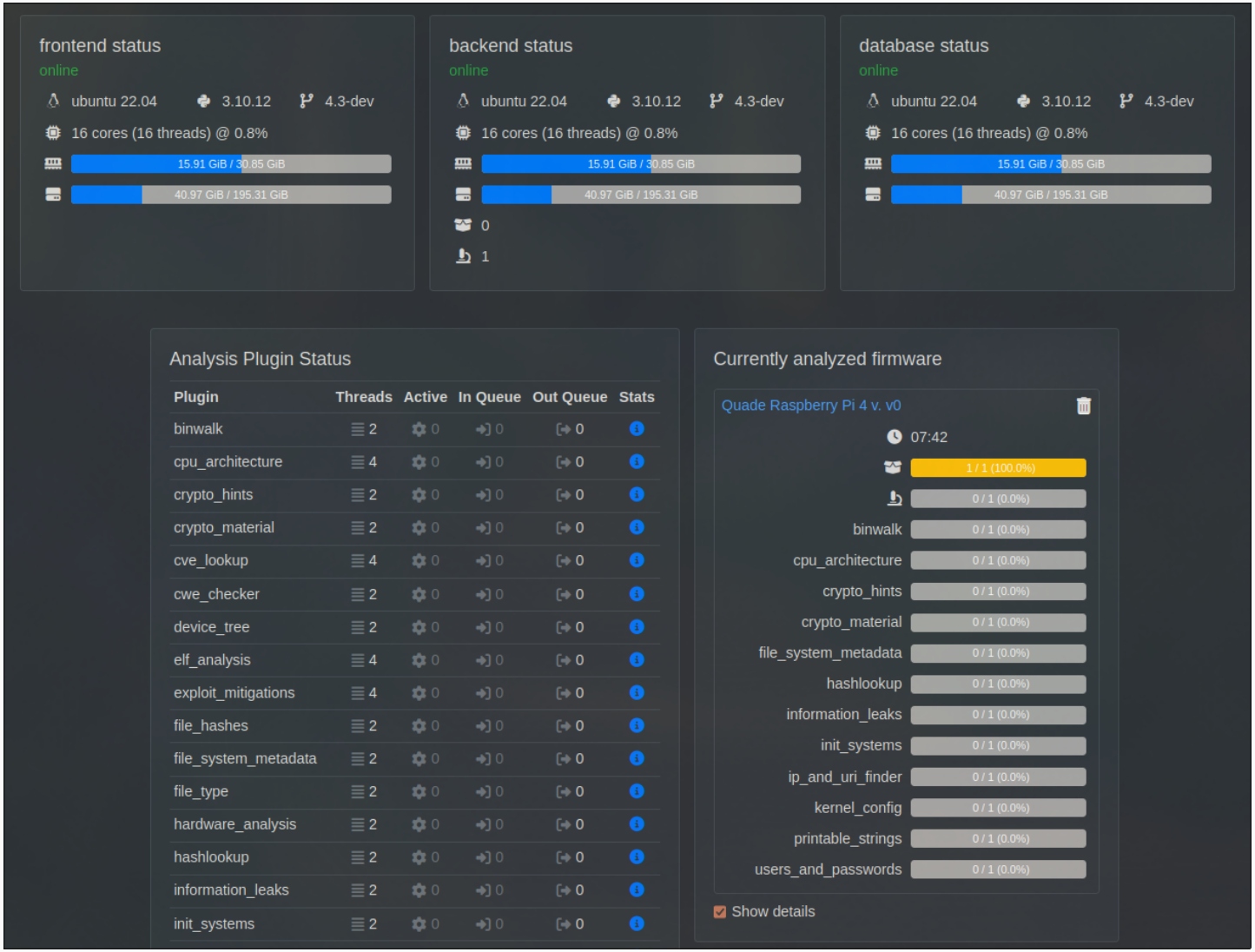

Figure 6: FACT

FACT (3)

Vorteile:

- Simples Webfrontend
- Strukturierte Übersicht aller Firmwarebestandteile
- Container-Umgebung welche “überall” läuft
- Perfekt für Massenanalysen

Nachteile:

- Hoher Ressourcenbedarf (CPU und RAM)
- Probleme bei der Auswertung von komplexen oder nicht-trivialen Images

Autopsy (1)

Autopsy ist ein grafisches Open-Source-Framework zur digitalforensischen Analyse von Datenträgern und Images.

- Basiert auf dem Kommandozeilen-Toolkit `Sleuth Kit (TSK)`
- Analyse von Partitionen, Dateisystemen und gelöschten Dateien
- Auswertung von Metadaten, Logs, Zeitstempeln, ...
- Extraktion und Kategorisierung verdächtiger Dateien

Architektur:

- Java-basiertes Desktop-GUI-Frontend
- Native Integration von TSK-Komponenten (via Java-Bindings)
- Unterstützung von Plugins zur Erweiterung

Autopsy (2)

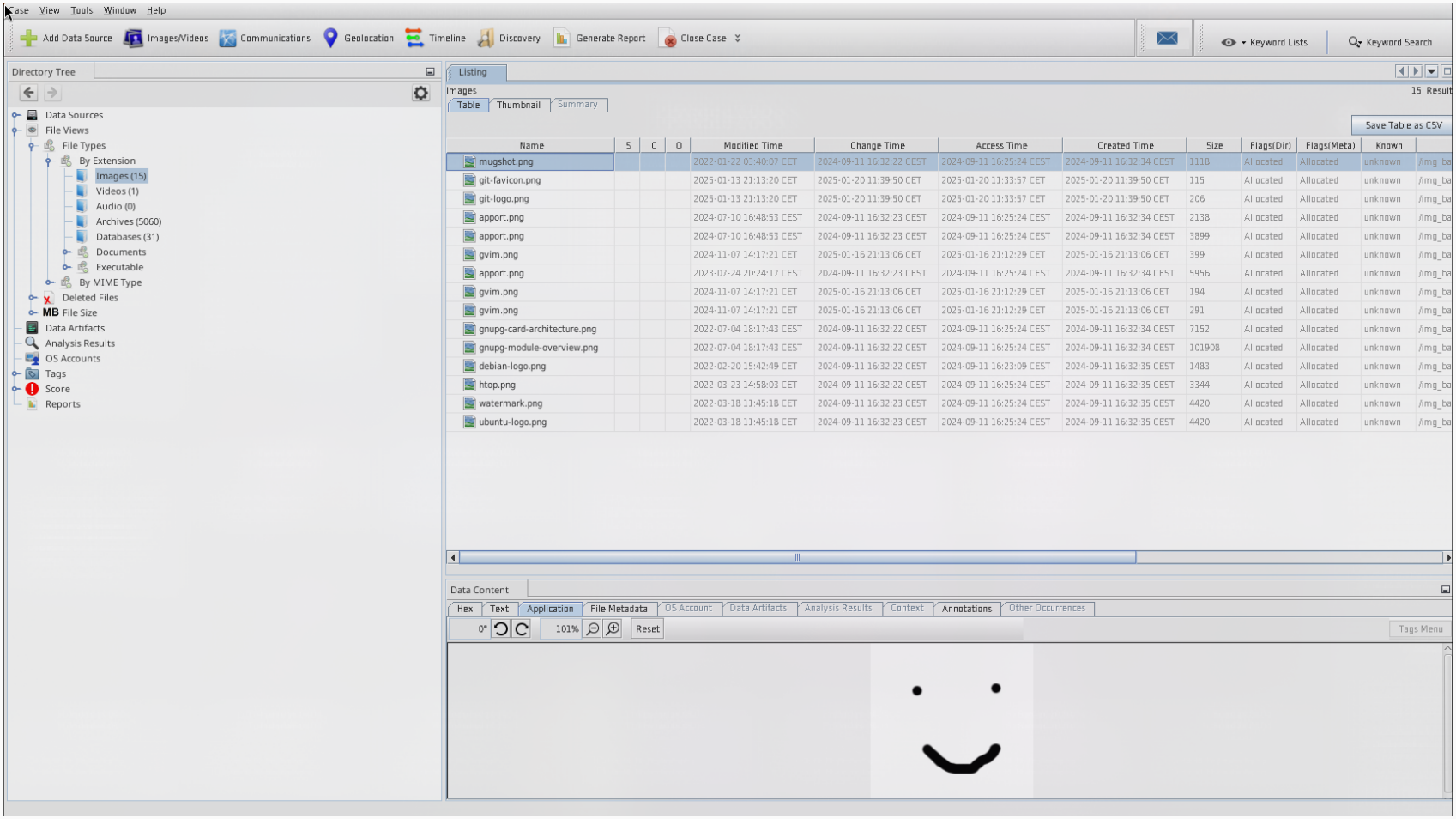

Figure 7: Autopsy

Autopsy (3)

Vorteile:

- Intuitives GUI zur Navigation durch komplexe Dateisysteme
- Ideal zur forensischen Analyse bereits extrahierter Root-Dateisysteme
- Integrierte Tools zur Timeline-, Hash- und Keyword-Analyse
- Teil von Kali Linux

Nachteile:

- Fokus liegt auf klassischen Datenträgerabbildern (NTFS, ext4)
- Ungeeignet für die Analyse roher Firmware-Images oder komprimierter Filesysteme
- Falls kein Installer vorliegt müssen die TSK Java-Bindings "per Hand" erzeugt werden

Vielen Dank für Ihre Aufmerksamkeit!