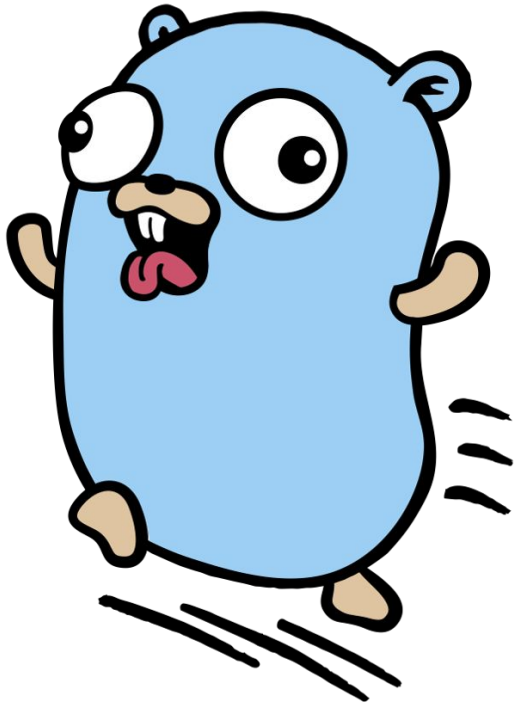
gokrazy

GPN17

Michael Stapelberg



Agenda

- Motivation
- Übersicht (mit Demo)
- Hardware und Cross-Compilation
- Partitionen, Dateisystem und Updates
- Firmware, Kernel und deren Updates
- Anwendungsfälle
- Fragen?

Motivation

- Raspberry Pi erscheint 2012 während Hausbus-Projekt im RaumZeitLabor

  ⟶ Automatisierung grundsätzlich nur noch in Hochsprachen auf Raspberry Pis

- Mein Ansatz für reproduzierbare read-only-Images: eigenes build script

- Updates erfordern große Blöcke Zeit, viel Testen, Fehler sieht man ggf. erst später

  ⟶ `RZL image build 2013-11-10.01` bis 2017 im Einsatz

  ⟶ Sicherheitslücken sammelten sich über 4 Jahre an

Motivation: Leitfragen

1. Kann man die Angriffsfläche stark reduzieren?

2. Kann man Updates komplett automatisieren?

3. Kann man die Einrichtung eines Raspberry Pis drastisch vereinfachen?

Übersicht über gokrazy

- Projekt, um in Go geschriebene Programme auf Raspberry Pis zu betreiben

- Simples init-system in Go implementiert

  → stark integriert: mit Web Interface, logging und Update-mechanismus

- Minimales Userland: derzeit NTP- und DHCP-Client, Linux macht den Rest

Demo

- lokaler DHCP-Server vorbereitet
- gokr-packer, auf SD-Karte installieren
- Boot via serieller Konsole verifizieren

Unterstützte Hardware: Raspberry Pi 3

- Erste Version, die vom upstream Linux-Kernel unterstützt wird
- Ethernet-port: gokrazy kann derzeit kein WiFi
- Test-Matrix möglichst klein halten, damit das Projekt einfach wartbar bleibt

Randnotiz: cross-compilation in Go

- Kompilieren und installieren (host):

  `go install github.com/gokrazy/hello`

- cross-compilation (target):

  `GOARCH=arm64 GOOS=linux go install`

  `github.com/gokrazy/hello`

- Keine libc? `CGO_ENABLED=0`, fertig. `init=/hello` auf Raspberry Pi läuft.

Randnotiz: „init-system“ in Go

```
package main

import (
	"os/exec"
	"time"
)

func supervise(cmd *exec.Cmd) {
	for {
		cmd.Run()
		time.Sleep(1 * time.Second)
	}
}

func main() {
	go supervise(exec.Command("/hello"))
	// …
	select{}
}
```

Partitionen, Dateisystem (wie kommt Software auf den Pi)

- Raspberry Pi erwartet eine SD-Karte

  mit MS-DOS partition table

  und FAT-Dateisystem als 1. Partition, welche die Firmware enthält

- ```
  sudo fdisk /dev/sdb # …
  sudo mkfs.vfat /dev/sdb1
  sudo mount /dev/sdb1 /mnt/firmware
  sudo cp init hello vmlinuz *.txt firmware/* /mnt/firmware
  sudo umount /mnt/firmware
  ```
  ```

Partitionen, Dateisystem (2)

- `fdisk`, `mkfs` und filesystem/loop mounts

  unbequem zu automatisieren (Schnittstelle, cleanups)

  erfordern root-Rechte

  auf manchen Betriebssystemen (z.B. macOS) nicht vorhanden

  nicht in Go geschrieben ;-)

- Eigene Implementation!

Partitionen (3)

```
const invalidCHS = [3]byte{0xFE, 0xFF, 0xFF} // results in using the sector values instead

func writePartitionTable(w io.Writer) error {
	for _, v := range []interface{}{
		[446]byte{},            // boot machine code (empty, unused)

		// partition 1
		byte(0x80),             // active partition (“boot flag”)
		invalidCHS,
		byte(0xc),              // FAT
		invalidCHS,
		uint32(8192),           // start at 8192 sectors
		uint32(100 * MB / 512), // 100MB in size

		// partition 2-4 omitted from slide for space reasons

		uint16(0xAA55),         // signature
	} {
		if err := binary.Write(w, binary.LittleEndian, v); err != nil { return err }
	}
	return nil
}
```

FAT16B-Dateisystem (4)

- < 500 Zeilen, nur 8.3-Dateinamen wegen Patenten

- boot sector, file allocation table (FAT), root directory, data area

- padding auf sector size (512) und cluster size (z.B. 4 * sector size)

  → bestimmt overhead und maximale Dateisystem-Größe (127 MB bei uns)

Einschub: Updates

Partition	Größe	Zweck	Dateisystem
1	100 MB	boot (kernel/firmware)	FAT16B
2	500 MB	root2 (gokrazy/apps)	FAT16B*
3	500 MB	root3 (gokrazy/apps)	FAT16B*
4	Rest	persistente Daten	z.B. ext4

- A/B-Mechanismus für Root-Dateisystem (boot wird eingelesen, root wird benutzt)
- minimaler FAT16B-reader, welcher den Ort der cmdline.txt zurückgibt

Firmware

- lebt in github.com/gokrazy/firmware

- `boot/*.{elf,bin,dat}` von github.com/raspberrypi/firmware

- proprietäre Binaries, die wir einfach kopieren

  → freie Firmware in Arbeit: github.com/christinaa/rpi-open-firmware

  (derzeit noch ohne USB, DMA, Ethernet)

Kernel

- lebt in github.com/gokrazy/kernel

- build-Programm (in Go), welches den Kernel reproduzierbar für arm64 kompiliert

- läuft in einem Docker-container (fixe Umgebung, einfach auf z.B. travis nutzbar)

  (Debian stretch setup: `apt install crossbuild-essential-arm64`)

- Resultat: vmlinuz (kernel), rpi-3-b.dtb (device tree), keine initrd

Kernel (2)

```
func compile() error {
	if err := exec.Command("make", "ARCH=arm64", "defconfig").Run(); err != nil {
		return fmt.Errorf("make defconfig: %v", err)
	}
	// omitted from slides for space reasons: modify .config
	if err := exec.Command("make", "ARCH=arm64", "olddefconfig").Run(); err != nil {
		return fmt.Errorf("make olddefconfig: %v", err)
	}

	make := exec.Command("make", "Image.gz", "dtbs", "-j8")
	make.Env = append(os.Environ(),
		"ARCH=arm64",
		"CROSS_COMPILE=aarch64-linux-gnu-",
		"KBUILD_BUILD_USER=gokrazy",
		"KBUILD_BUILD_HOST=docker",
		"KBUILD_BUILD_TIMESTAMP=Wed Mar  1 20:57:29 UTC 2017",
	)
	make.Stdout = os.Stdout
	make.Stderr = os.Stderr
	return make.Run()
}
```

Firmware- und Kernel-Updates

1. travis-cronjobs (1, 2) machen Pull Requests (1, 2) mit neuer Upstream-Version auf

2. Pull Requests werden auf travis gebaut (firmware-download, kernel-build)

3. Resultat wird in den Pull Request eingearbeitet

4. Pull Requests werden auf travis getestet (gokr-boot, bakery) und gemerged.

Zusammenfassung

- Angriffsfläche stark reduziert: 4 Komponenten (+ Anwendungen), alles in Go
- Updates komplett automatisch
- 1 Befehl für Installation auf SD-Karte, für Image-Erstellung oder Netzwerk-Updates

Anwendungsfall (1): Media-automatisierung

- „avr-x1100w“ steuert den gleichnamigen A/V Receiver

- überwacht Computer (ping) und Chromecasts

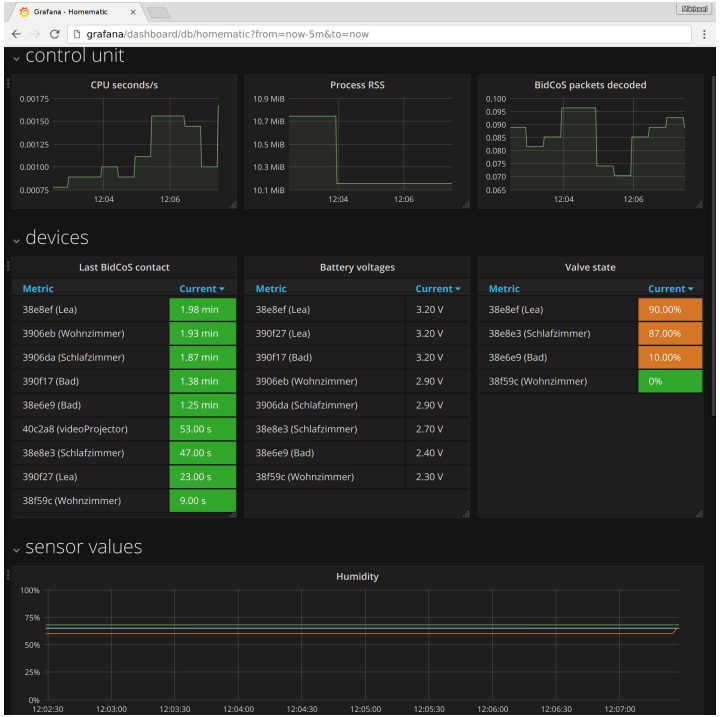
  → beim Abspielen werden Geräte automatisch umgeschaltet

- ähnlich wie HDMI-CEC, aber auch für Geräte, die über TOSLINK angebunden sind

Anwendungsfall (2): Hausautomatisierung

- hmgo ist ein Ersatz für HomeMatic CCUs
- Minimale APIs, keine Konfigurierbarkeit, kein UI.

  Mein Anwendungsfall ist hard-coded.
- Monitoring/Auswertung via Prometheus



Anwendungsfall (3): Backups

- dornröschen weckt Rechner und Network Storages zeitgesteuert auf

- Nach dem Aufwecken wird ein Backup via SSH angestoßen

  (Backup via rsync, was auf dem jeweiligen Host läuft)

Anwendungsfall (4): scan2drive

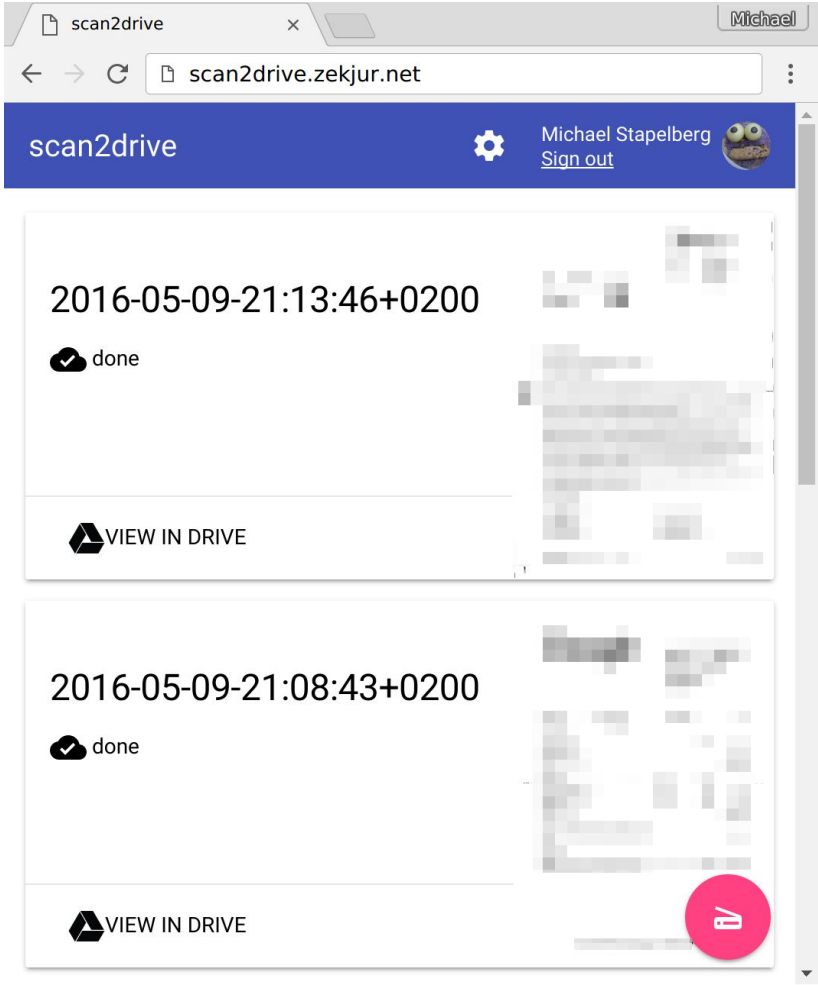
- scan2drive nutzt den Fujitsu ScanSnap iX500.

  Es scannt auf Knopfdruck Dokumente ein,

  verschönert/verkleinert die JPGs in ein PDF,

  lädt das PDF auf Google Drive hoch

  → Volltextsuche via Google Drive



Fragen?

- Danke für die Aufmerksamkeit!
- Mehr Infos: https://gokrazy.org/
- Fragen?
- Feedback zum Vortrag