My key takeaways from my 5-year experience of developing and maintaining two open source projects aimed at automating iOS devices

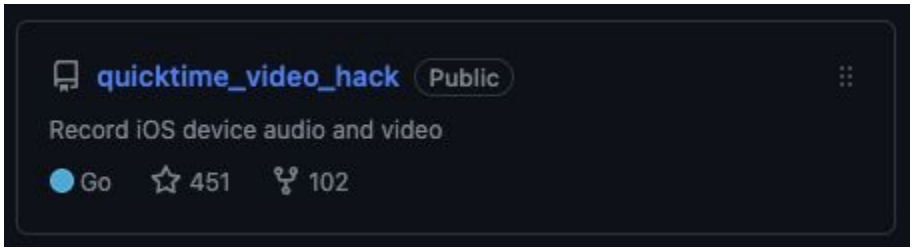
and why working with the Gopher was a good choice!

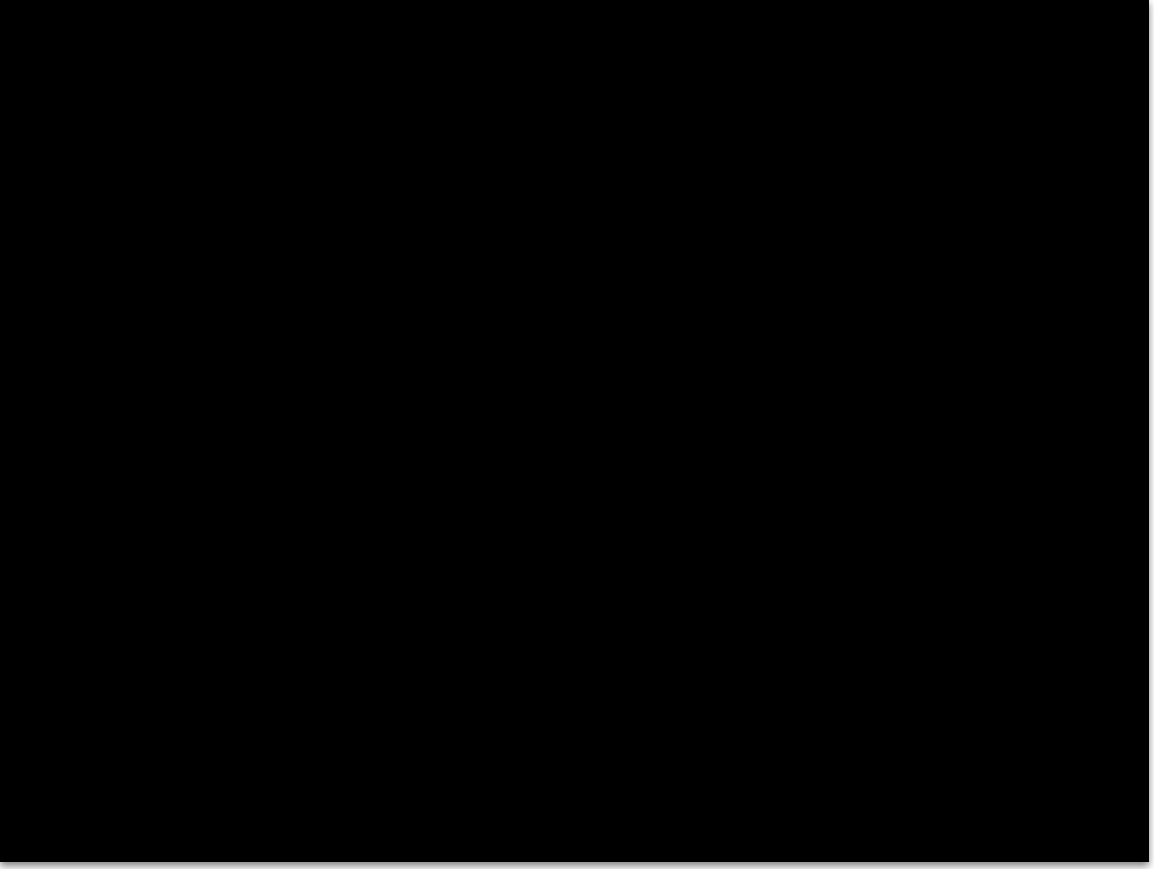
Let’s watch 2 minutes of a demo to give you all some context

https://www.youtube.com/watch?v=aqM-g01qP2c

1. The story of QVH

quicktime_video_hack, or why I suck at naming projects 😅😅





Mac OS X allows you to mirror iOS video and audio with Quicktime. And I wanted to build the same, but on Linux

BUT HOW TO SOLVE? 😱

1. let's google it ☝️😌

2. found video on youtube 🥳

3. found discussions online without solutions 😑

4. no source code 😭

5. how hard can it be? 🤨

Two ways of finding out how these features work

1. Try and understand from disassembled code

Disassemble the binary and try to understand it using static analysis and a debugger

```
int sub_10001852b(int arg0, int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8) {
    var_24 = arg8;
    var_16 = arg7;
    var_8 = arg6;
    r15 = arg2;
    r14 = arg1;
    var_38 = arg5;
    var_2C = arg0;
    r13 = arg3;
    rbx = arg4;
    if (*qword_100021cc8 != 0xffffffffffffffff) {
            dispatch_once(qword_100021cc8, ^ {/* block implemented at sub_100011220 */ } );
    }
    r12 = *qword_100021cc0;
    var_48 = CFStringCreateWithCharacters(**_kCFAllocatorDefault, r14, r15);
    *(int8_t *)(&var_48 + 0x8) = 0x1;
    var_58 = r13;
    *(int32_t *)(&var_58 + 0x8) = rbx;
    sub_100012324(r12, var_2C, &var_48, &var_58, var_38, var_8, var_16, var_24);
    sub_10000e8d8(&var_48);
    rax = rax;
    return rax;
}
```

That’s pretty hard 🤔

Two ways of finding out how these features work

Eavesdropping on USB is luckily quite easy nowadays on Mac OS X

```
sudo ifconfig XHC20 up
```

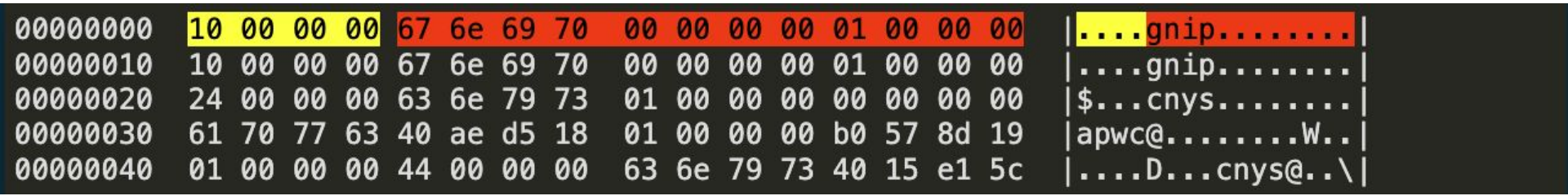
Using Wireshark I extract a Hexdump

But what does it all mean?

```
00000000  10 00 00 00 67 6e 69 70  00 00 00 00 01 00 00 00  |....gnip........|
00000010  10 00 00 00 67 6e 69 70  00 00 00 00 01 00 00 00  |....gnip........|
00000020  24 00 00 00 63 6e 79 73  01 00 00 00 00 00 00 00  |$...cnys........|
00000030  61 70 77 63 40 ae d5 18  01 00 00 00 b0 57 8d 19  |apwc@........W..|
00000040  01 00 00 00 44 00 00 00  63 6e 79 73 40 15 e1 5c  |....D...cnys@..\|
00000050  fb 7f 00 00 74 6d 66 61  90 4d af 18 01 00 00 00  |....tmfa.M......|
00000060  00 00 00 00 00 70 e7 40  6d 63 70 6c 4c 00 00 00  |.....p.@mcplL...|
00000070  04 00 00 00 01 00 00 00  04 00 00 00 02 00 00 00  |................|
00000080
```

How do Messages work when sent over Streams?

1. Fixed length
2. Delimiter-based messages
3. 4 byte int containing length + payload of that length



The strings are a good starting point but what is a gnip?

Endianness! It’s ping not gnip! Suddenly it all makes sense!

```
ASYN              uint32 = 0x6173796E //nysa - asyn
FEED              uint32 = 0x66656564 //deef - feed
RELS              uint32 = 0x72656C73
HPD1              uint32 = 0x68706431 //hpd1 - 1dph |
HPA1              uint32 = 0x68706131 //hpa1 - 1aph |
NEED              uint32 = 0x6E656564 //need - deen
EAT               uint32 = 0x65617421 //contains audio
KeyValuePairMagic uint32 = 0x6B657976 //keyv - vyek
StringKey         uint32 = 0x7374726B //strk - krts
IntKey            uint32 = 0x6964786B //idxk - kxdi
BooleanValueMagic uint32 = 0x62756C76 //bulv - vlub
DictionaryMagic   uint32 = 0x64696374 //dict - tcid
DataValueMagic    uint32 = 0x64617476 //datv - vtad
```

Working through the Hex..

Here you will find sbuf, which is a serialized CMSampleBuffer instance from Apple's CoreMedia Framework. It contains raw h264 units in sdat!

```
00000000  d7 65 01 00 6e 79 73 61  60 2f c3 5c fb 7f 00 00  |.e..nysa`/.\....|
00000010  64 65 65 66 c3 65 01 00  66 75 62 73 20 00 00 00  |deef.e..fubs ...|
00000020  73 74 70 6f 68 54 8d 40  3b 57 00 00 00 ca 9a 3b  |stpohT.@;W.....;|
00000030  01 00 00 00 00 00 00 00  00 00 00 00 50 00 00 00  |............P...|
00000040  61 69 74 73 01 00 00 00  00 00 00 00 3c 00 00 00  |aits........<...|
00000050  01 00 00 00 00 00 00 00  00 00 00 00 68 54 8d 40  |............hT.@|
00000060  3b 57 00 00 00 ca 9a 3b  01 00 00 00 00 00 00 00  |;W.....;........|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 86 62 01 00  |.............b..|
00000090  74 61 64 73 00 00 00 1e  06 05 1a 47 56 4a dc 5c  |tads.......GVJ.\|
000000a0  4c 43 3f 94 ef c5 11 3c  d1 43 a8 01 ff cc cc ff  |LC?....<.C......|
000000b0  02 01 31 2d 00 80 00 01  62 58 25 b8 20 07 ff f0  |..1-....bX%. ...|
000000c0  84 35 2d 02 55 ff 03 5b  4d cd b6 1a 16 09 f1 5d  |.5-.U..[M......]|
000000d0  46 bf ea 3e 4d 0e 0c e4  6d 84 5a 00 00 03 00 00  |F..>M...m.Z.....|
000000e0  03 00 00 03 00 01 42 bc  53 80 bf d1 d8 3b 47 e0  |......B.S....;G.|
000000f0  21 61 98 83 c3 d1 62 92  4f 0f 67 7f 4a 6b 10 6a  |!a....b.O.g.Jk.j|
```

Finally, create a video

A nice example for a delimiter based protocol, writing h264 raw NaLus like this, will create a playable video file!

```
var delimiter = []byte{00, 00, 00, 01}

func (avfw AVFileWriter) writeNalu(naluBytes []byte) error {
        _, err := avfw.h264FileWriter.Write(delimiter)
        if err != nil {
                return err
        }
        _, err = avfw.h264FileWriter.Write(naluBytes)
        if err != nil {
                return err
        }
        return nil
}
```

Why is Golang great for this?

Accessing USB devices is very easy with the gousb package

```
ctx := gousb.NewContext()
devices, err := ctx.OpenDevices(func(desc *gousb.DeviceDesc) bool {
        // this function is called for every device present.
        // Returning true means the device should be opened.
        return validDeviceChecker(desc)
})
device := devices[0]
conf, _ := device.Config(configIndex)
iface, _ := conf.Interface(confNum, altSettingIndex)
inEndpoint, _ := iface.InEndpoint(grabInboundBulkEndpoint(iface.Setting))
outEndpoint, _ := iface.OutEndpoint(grabOutboundBulkEndpoint(iface.Setting))
stream, _ := inEndpoint.NewStream(4096, 5)
buffer := make([]byte, 65536)
stream.Read(buffer)
//do things with the buffer contents...
outEndpoint.Write([]byte{1,2,3,4})
```

Why is Golang great for this?

Unlike Java, there are unsigned ints! That makes network and protocol coding much nicer.

```
//you have unsigned ints for every byte size,
//makes the java developer in me wanna cry out of happiness
var unsigned_one_byte_integer uint8 = 3
var unsigned_two_byte_integer uint16 = 6
var unsigned_four_byte_integer uint32 = 12
var unsigned_eight_byte_integer uint64 = 24

//also, converting primitives back and forth
//is elegant and simple
var some_float float64 = 0.5
var float_as_uint64 uint64
float_as_uint64 = math.Float64bits(some_float)
binary.LittleEndian.PutUint64(someByteArray, float_as_uint64)
```

Why is Golang great for this?

Byte Slices are a true blessing when building codecs

```
responseBytes := make([]byte, 24)
binary.LittleEndian.PutUint32(responseBytes, 24)
binary.LittleEndian.PutUint32(responseBytes[4:], ReplyPacketMagic)
binary.LittleEndian.PutUint64(responseBytes[8:], sp.CorrelationID)
binary.LittleEndian.PutUint32(responseBytes[16:], 0)
binary.LittleEndian.PutUint32(responseBytes[20:], 0)
//or
responseBytes := make([]byte, 60)
length := writePayload(responseBytes[24:])
writeHeader(responseBytes[:24], length)
```

Why is Golang great for this?

You can even write structs directly to byte streams!

```
type CMTime struct {
        CMTimeValue uint64
        CMTimeScale uint32
        CMTimeFlags uint32
        CMTimeEpoch uint64
}
func NewCMTimeFromBytes(data []byte) (CMTime, error) {
        r := bytes.NewReader(data)
        var cmTime CMTime
        err := binary.Read(r, binary.LittleEndian, &cmTime)
        if err != nil {
                return cmTime, err
        }
        return cmTime, nil
}
```

Reverse Engineering Makes You a Better Engineer!

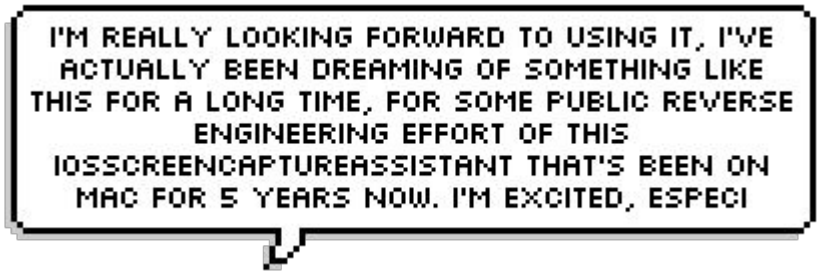
Make theories on how "they" built it, and test them one by one

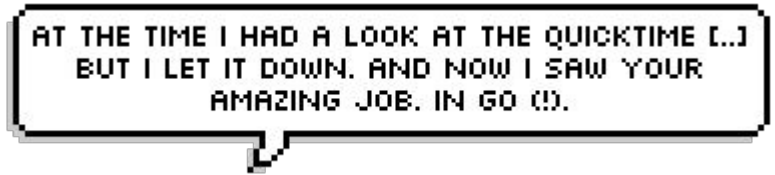
Writing clean, unit tested application code without knowing the end result

Learn many cool new things like: networking basics, h264, USB coding

Low Level: There is no magic

+You get a lot of Love from people 🥰





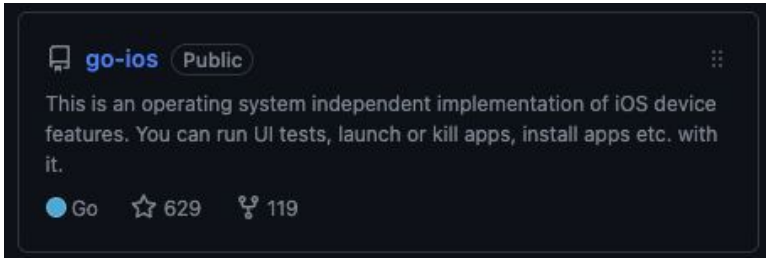
2. Go-iOS

It was written in Go, and it does things with iOS devices, yes I suck at naming things still 😅

Some stats and a cute logo

Sponsored by big testing companies,

used by many more



Open Source Lib Design Considerations

Written in Golang → AOT compiled binary, great for using it from other tools

The CLI output is JSON first for easy parsing

Uses DocOpt, all in one binary command

Just run “npm install -g go-ios” and you’re good to go

I like convenience, many commands automagically configure things or make them very easy

Open Source Lib Design Considerations

Comes with modules and the CLI code to explain how to use them, importing it into your Go programs is easy

Comes with a powerful debug proxy tool

MIT License, if you want to build cool stuff with it like automated- iOS device configuring raspberry pi’s.. well you can! And I am happy to help

Pitfalls

Golang library dev pitfalls I learned about the hard way

Golang is not Java - Daniel’s turn

I wrote a lot of Netty (Async Java Networking) before, so I thought this was a good idea:

1. Read messages from the device in a for loop in a separate go routine, using a custom codec
2. Set up a channel that would allow to receive messages
3. Allows async code, but was totally unnecessary and overengineered
4. Unfortunately sometimes, you need to change codecs on the fly..

This is not Java - Daniel’s turn

Which resulted in terrible code like this:

```
func (conn *DeviceConnection) sendForProtocolUpgrade(muxConnection *MuxConnection,
message interface{}, newCodec Codec) []byte {
        log.Debug("Protocol update to ", reflect.TypeOf(newCodec), " on ", &conn.c)
        conn.stopReadingAfterNextMessage()
        conn.send(message)
        responseBytes := <-muxConnection.ResponseChannel
        conn.activeCodec = newCodec
        conn.startReading()
        return responseBytes
}
```

This is not Java - Daniel’s turn

I later realized, synchronous code is totally fine and with the Reader interface I could solve this problem much more elegantly:

```
// ReadMessage grabs the next LockDown Message using the PlistDecoder from the underlying
// DeviceConnection and returns the Plist as a byte slice.
func (lockDownConn *LockDownConnection) ReadMessage() ([]byte, error) {
        reader := lockDownConn.deviceConnection.Reader()
        resp, err := lockDownConn.plistCodec.Decode(reader)
        if err != nil {
                return make([]byte, 0), err
        }
        return resp, err
}
```

A Fatal misconception

log.Fatal() sounds like a really nice way to fail when, well things fail and cannot be recovered. However:

```
func (logger *Logger) Fatal(args ...interface{}) {
    logger.Log(FatalLevel, args...)
    logger.Exit( code: 1)
}
```

- defer does not work 😱
- test clean up does not work 😱 (can be a big deal with stateful stuff, like physical devices)
- I even had this in library code, not just tests 🙈 🙊🙈 🙊🙈 🙊🙈 🙊🙈 🙊

😭 CGO 😭

CGO can be good with stable, maintained libs like gousb - in any other case

🔥🔥🔥🔥 ***Welcome - to - Hell*** 🔥🔥🔥🔥

I tried to use Gstreamer, created a fork of a fork to make it work

Memory leaks, segfaults, weird errors.

panic() & recover() are not global 😣 😖 😫 😩

you must add recover in *every* goroutine, there is no bubbling up of panics like in f.ex. NodeJS

go-ios contains A LOT of type casts and map access, this made some users really sad

How do you actually handle errors?

```
func (conn *Connection) Push(srcPath, dstPath string) error {   wangchao +1
    ret, _ := ios.PathExists(srcPath)
    if !ret : fmt.Errorf("%s: no such file.", srcPath) ↗

    f, err := os.Open(srcPath)
    if err != nil : err ↗
    defer f.Close()

    if fileInfo, _ := conn.Stat(dstPath); fileInfo != nil {
        if fileInfo.IsDir() {
            dstPath = path.Join(dstPath, filepath.Base(srcPath))
        }
    }

    return conn.WriteToFile(f, dstPath)
}
```

How do you actually handle errors?

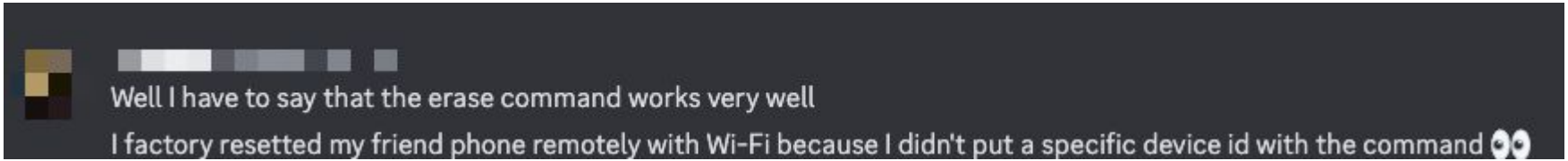
This is how I will do it from now on as it gives me something like a stracktrace. But I actually don’t know if this a good way to solve it. If you can share your views on how to do great error handling in go, please do.

```
func (conn *Connection) Push(srcPath, dstPath string) error {  wangchao +1 *
    ret, _ := ios.PathExists(srcPath)
    if !ret : fmt.Errorf("%s: no such file.", srcPath)

    f, err := os.Open(srcPath)
    if err != nil {
        return fmt.Errorf( format: "connection.push: failed with err: %w", err)
    }
    defer f.Close()
```

Make sure people don’t delete their friend’s phones 😅

I am still sorry about this one. Always keep in mind that folks might actually *use* your library, so make good design decisions and ensure it is safe to use.



Being an Open source maintainer is awesome and you can do it too!

Practical tips

Build something useful to learn a new language and open source it, don’t just follow tutorials and create uninteresting stuff on Github

Build a community and be accessible

Be friendly and kind

Practical tips

Look for sponsors

Look for contributors

If you decide to go with MIT, people will copy your code, companies will use it without any attribution or paying for it. That’s just the way things are :-)

Try growth hacks! It’s fun!

Create content! (I need to do that more)

Practical tips

I work on this at night when the kids are sleeping

I work on this when my wife is driving and I am sitting next to her

I write code at night

It’s hard and sometimes I need a break to not burn out

→ ***Do take breaks, accept your personal limits***

What I get out of it and what it taught me

Prioritization& Product Thinking!

Seeing other people contribute is absolutely beautiful

I built something that people actually use

Working on something long term, where your decisions matter

I met incredibly cool people and made great friends

and finally… **THE USE CASES**!

Closing comments

- Next up: iOS 17 support. It’s tough, but I got some help now.
- Next up: the REST API will be finished you can use it from other languages more easily
- Next up: Some more videos and blog posts to make it easier to use it

Call for contribution: Are you interested in fiddling with devices and low level code? Contributing to open source? Or maybe in writing great docs, recording videos, helping the community out and similar? I can definitely need all the help I can get! Seek me out later or reach out on Discord please.