u-root: A Go-based, Firmware Embeddable Root File System with On-Demand Compilation

Ron Minnich, Gan-shun Lim, Ryan OLeary, Chris Koch, Google

Andrey Mirtchovski Cisco

Outline

- Go in 60 seconds
- What u-root is
- How it all works
- Using Go ast package to transform Go
- Where we’re going

Go in 60 seconds

- New language from Google, released 2009
- Creators include Ken, Rob, Russ, Griesemer
- Not Object Oriented
  - By design, not ignorance
- Designed for systems programming tasks
  - And really good at that
- My main user-mode language since 2010
- Addictive

Go in 60 seconds:goo.gl/dIJrYG

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

var a struct {
        i, j int
}
```

- Every file has a package
- Must import packages you use
- Declare ‘a’ as an anon struct

Go in 60 seconds

Could also say:

```
type b struct {
    I, j int
}
var  a b
```

- Note declarations are Pascal-style, not C style!
- “The type syntax for C is essentially unparsable.” - Rob Pike

Go in 60 seconds: goo.gl/dlJrYG

```
func init() {
	a.i = 2
}

func main() {
	b := 3
	fmt.Printf("a is %v, b is %v\n", a, b)
}
```

- init() is run before main
- You can have many init() functions
- b is declared and set
- %v figures out type

Could also say ...

fmt.Printf(“%d”, b)

Package example https://goo.gl/X2SqyZ

```
// You can edit this code!
// Click here and start typing.
package hi

var (
	internal int
	Exported int
)
```

- variables/functions starting in lowercase are not visible outside package; those starting in Uppercase are
- No export/public keyword

Package: https://goo.gl/X2SqyZ

```
func youCanNotCallFromOutside() {
    fmt.Println("hi")
}
func YouCanCallFromOutside() {
    fmt.Println("hi")
}
```

First class functions:goo.gl/pP4FcJ

```
package main

import "fmt"

var c = func(s string) {fmt.Println("hi", s)}
func main() {
        p := fmt.Println
        p("Hello, 世界")
        c(" there")
}
```

Go in 60 seconds

- Compiler is really fast (originally based on Plan 9 C toolchain)
- V 1.2 was fastest; currently at 1.9, rewritten in Go, is still quite fast
- Compile all of u-root, including external packages, in under 15 seconds
- Package syntax makes finding all imports easy

u-root

- Go-based rootfs
  - Commands/packages written in Go
  - In one mode, MAX, compiled on demand
- 1 or 4 pre-built binaries:
  - /init
  - Go toolchain -- if compiling on demand
- Type a command, e.g. rush (shell)
  - rush and its packages are compiled to /ubin and run
  - Compilation is minimal and fast (½ second)

Key idea: $PATH drives actions

- PATH=/bin:/ubin:/buildbin
  - /bin is *usually* empty
  - /ubin is *initially* empty
- /buildbin has symlinks to an *installcommand*
- First time you type rush: found in /buildbin
  - Symlink in /buildbin: rush -> installcommand
  - Installcommand runs, builds argv[0] into /ubin
    - Execs /ubin/rush
- Next time you type rush, you run /ubin/rush

Installcommand is built on boot

- Init builds installcommand in /buildbin
- For each d in /src/github.com/u-root/u-root/cmds/*, init creates /buildbin/d -> /buildbin/installcommand
- init forks and execs rush
  - which may be compiled by the installer and run
- init: 206 lines

“U” is for “Universal”

- Single root device for all Go targets
- New architecture requires only 4 binaries
- For multi-architecture root, proper (re)arrangement of paths is needed
  - E.g., /init -> /linux_<arch>/init

Variations on u-root for embedded

- Not everyone wants source in FLASH
- Some FLASH parts are small
- Hence the root image can take many forms
- But source code never changes
  - I.e. no specialized source code for embedded

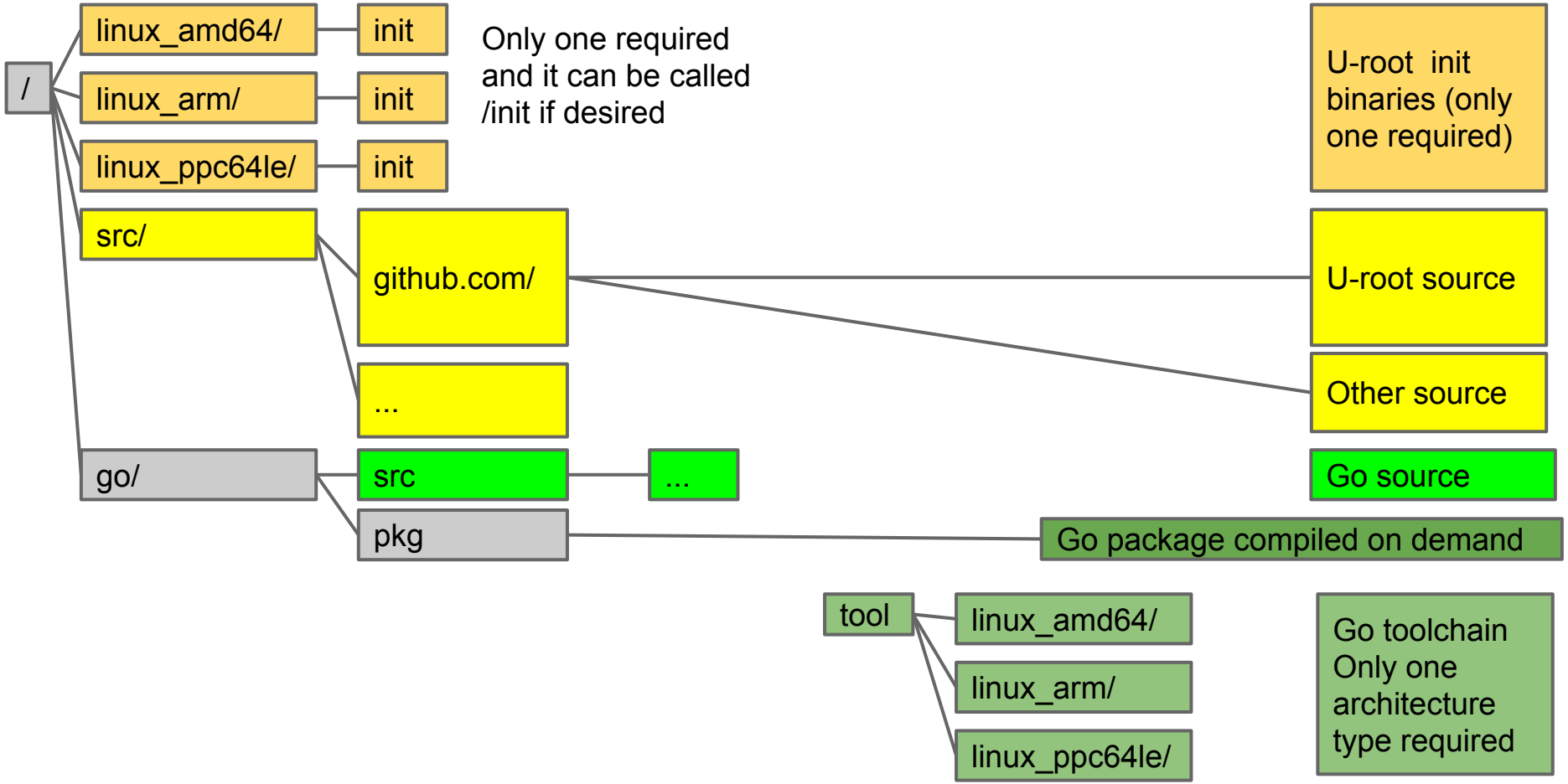
Variations of u-root

4 binaries per architecture, all commands in source form, dynamic compilation, multiple architectures in one root device	Post-boot model -- i.e. local disk, nfsroot, etc.	MAX
More than 4 binaries per architecture: some/all commands precompiled, dynamic compilation, multiple architectures in one root image	Post-boot model where faster boot is required	
4 binaries, all commands in source form, dynamic compilation, one architecture	Pre- or Post- boot model: u-root installed in firmware or local device	
All commands built into one binary which forks and execs each time	Usually firmware but also netboot of “kexec” image	MIN

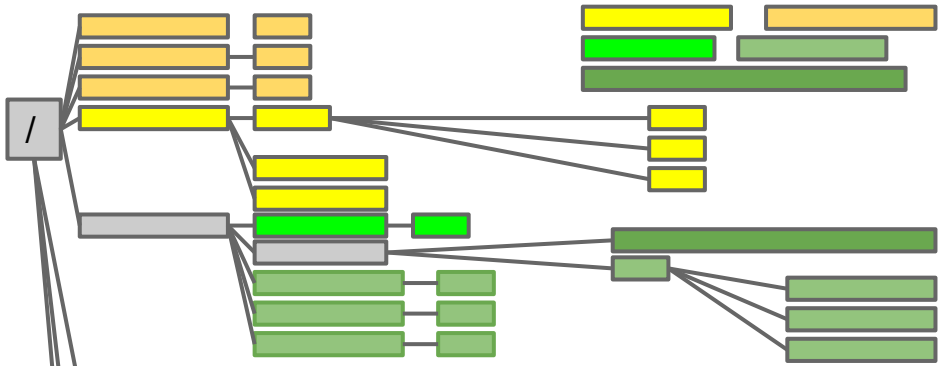
A deeper look at u-root “MAX”

- Standard kernel
- four Go binaries *per architecture**
  - init/build binary (part of u-root, written in Go)
    - Merged-in minimized go build tool
  - Compile, asm, link
- All *required* Go package **source**
- u-root source for basic commands
- in 5.9M (compressed of course! :-)

Root structure at boot



Init builds directories, mounts, ...



buildbin/

installcommand

rush -> installcommand
cat -> installcommand
...

ubin/

create etc/, dev/, proc/
mknod, mount, create any needed files (e.g. resolv.conf)

installer binary

Directory of symlinks built by init

Init creates required device nodes, mount points, and mounts

Init tasks

- /ubin is empty, mount tmpfs on it
- /buildbin is initialized by init with symlinks to a binary which builds commands in /bin
- PATH=/go/bin:/bin:/ubin:/buildbin
- create /dev, /proc, /etc
- Create inodes in /dev
- mount procfs
- Create minimal /etc/resolv.conv

Running first sh (rush)

- Init forks and execs rush
- If rush is not in /ubin, falls to /buildbin/rush (symlink->installcommand) runs
- /buildbin/installcommand directs go to build rush, and then execs /ubin/rush
- And you have a shell prompt
- From rush, same flow for other programs

Using Go to write more Go

- For scripting
- For dynamically creating shells with builtins
- For creating small memory pre-compiled versions of u-root (“busybox mode”)

Script for ip link command

```
run { ifaces, _ := net.Interfaces()
    for _, v := range ifaces {
        addrs, _ := v.Addrs()
        fmt.Printf("%v has %v", v, addrs)
    } }
```

- Result:

  ip: {1 1500 lo up|loopback} has [127.0.0.1/8 ::1/128]
  ip: {5 1500 eth0 fa:42:2c:d4:0e:01 up|broadcast} has [172.17.0.2/16 fe80::f842:2cff:fed4:e01/64]

- But it’s not really a program … how’s that work?

‘Run’ command rewrites fragment and uses the go import package

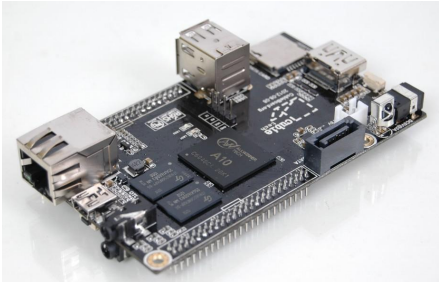
- run reads the program
  - If the first char is ‘{‘, assumes it is a fragment and wraps ‘package main’ and ‘func main()’ boiler plate
- Import uses the Go Abstract Syntax Tree (ast) package:
  - Parses a program
  - Finds package usage
  - Inserts go “import” statements

The result

- run program builds and runs the code
- Uses Go to write new Go

```
package main
import “net”
import “fmt”
func main(){
    ifaces, _ := net.Interfaces()
        for _, v := range ifaces {
                addrs, _ := v.Addrs()
                fmt.Printf("%v has %v", v, addrs)
        }
  }
```

Taking rewriting further



- Request for single-binary version of u-root for Cubieboard
  - Allwinner A10 --> not very fast
- Wanted to compile all u-root programs into one program

Taking rewriting further

- With the ast package, we can rewrite programs as packages, e.g. ls.go

```
package main

var x = flag.String(“l”, …)
func init() {...}
func main() {
}
```

```
package ls

var x = flag.String(“ls.l”, …)
func Init() {...}
func Main() {
}
```

- Combine all of u-root into one program
- Turning 65 programs into one: 10 seconds

What is all this good for?

- Building safer startup environments
- We can verify the root file system as in ChromeOS, which means we verify the compiler and source, so we know what we're running
- Much easier embedded root
- Security that comes from source-based root
- Knowing how things work

But I want bash!

- It’s ok!: tinycorelinux.net has it
- The tcz command installs tinycore packages
- tcz [-h host] [-p port] [-a arch] [-v version]
  - Defaults to tinycore repo, port 8080, x86_64, 5.1
- Type, e.g., tcz bash
- Will fetch bash and all its dependencies
- Once done, you type
- /usr/local/bin/bash (can be in persistent disk)

Where to get it

github.com/u-root/u-root

Instructions on

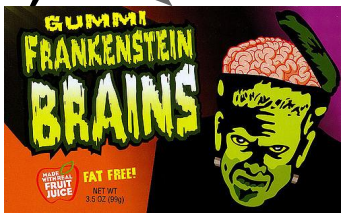
U-root.tk

What's this have to do with Google

- New project, NERF
- Non-Extensible Reduced Firmware
- Basic idea
  - Reduce functionality and extensibility of ME/UEFI
  - Replace what you lost with Linux
  - Arrange for PEI to call Linux instead of UEFI shell

Steps

- ME clean to make room
  - Minnow MAX: reduces 5M to 300K for ME (!!!)
  - Remove UEFI DXEs that provide net-, disk-, usb-, http-, file system- zero day support; keep UEFI shell
  - Remove all UEFI apps (no more ring 0 apps!)
- The fun bit: replace code in UEFI shell PE with a linux kernel (we love UEFITool)
- PEI then starts linux kernel
- Servers are back after 10 years!



Status

- Demonstrated on 4 motherboards
- Hope to have a single Go tool to do the job in a few months
- Looking for collaborators
- While we prefer coreboot-based systems we can use u-root on UEFI-based systems via NERF

Implications for coreboot

- NERF is like coreboot with ramstage removed
  - And linux replacing ramstage
  - Which is what LinuxBIOS was
- Did some experiments last year with replacing ramstage with Harvey-OS
- Plan to experiment this fall with Linux replacing coreboot ramstage

Further craziness

- RISCV has M-mode, which is like SMM
  - Problem: M-mode code, data visible to kernel
  - Fix: Make kernel provide M-mode blob
  - Hmm … on x86, let kernel provide SMM code, since we're replacing ramstage with kernel anyway?
- If kernel provides SMM blob we may close off a whole class of exploits
- And avoid friendly firmware writing garbage

Demo time (Thanks Marshall!)

Extra

HP FALCO 2-core chromebook, 4GiB

- First build of all packages for /bin/installcommand ~5s
  - runs 162 commands, builds many more files
- Subsequent commands are much faster because more packages are already built
- Date + 2 packages is 2 seconds
- Once built, it’s instantaneous (statically linked; in tmpfs!)

builtins? Need to go a bit deeper

- Normally builtins extend interpreted shell code
- u-root builtins extend the shell binary by recompiling the shell with new functions
- Builtin command is a superset of run
- First, let’s look at a shell builtin

Basic builtin(s)

```
builtin \
   hi ‘{ fmt.Printf(“hi\n”) }’ \
   there ‘{fmt.Println(“there”)}’
```

- Create a new shell with hi and there commands

Builtins combine script and rebuild

```
package main

import "errors"
import "os"

func init() {
	addBuiltIn("cd", cd)
}

func cd(cmd string, s []string) error {
	if len(s) != 1 {
		return errors.New("usage: cd one-path")
	}
	err := os.Chdir(s[0])
	return err
}
```

- This is the ‘cd’ builtin
- Lives in /src/sh
- When sh is built, it is extended with this builtin
- Create custom shells with built-ins that are Go code
- e.g. temporarily create purpose-built shell for init
- Eliminates init boiler-plate scripts

Customize the shell in a few steps

- create a unique tempdir
- copy shell source to it
- convert sets of Go fragments to the form in previous slide
- Create private name space with new /ubin
- mount --bind the tempdir over /src/cmds/rush/ and runs /ubin/rush
- You now have a new shell with a new builtin

The new shell

- Child shells will get the builtin
  - since they inherit the private name space
- Shells outside the private name space won’t see the new shell
- When first shell and kids exit, builtin is gone
- Custom builtins are far more efficient
  - Need a special purpose shell many times?
  - You can pay the cost once, not once per exec