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

Ron Minnich
Google

Andrey Mirtchovski
Cisco

Outline

- What u-root is
- How it all works
- Using Go ast package to transform Go
- Try it!
  - docker run --privileged -i -t rminnich/u-root:18 /bin/bash
  - cd /u-root && sh README

u-root

- Root file system in source form
- Type a command, e.g. date
  - Date and packages it needs get compiled to /bin
    - Usually in ramfs for performance
  - Newly compiled command is then run
  - Compilation is minimal and fast (½ second)
- Some variations on this theme (more later)
- All commands/packages are there as source

Wait! There has to be *some* binary ...

- And there is: /init (or /linux_arch/init)
- And of course the Go compiler: 4 binaries
- But the rest really is source
- On boot, /init compiles an installer
- Init then forks and execs sh
  - which is compiled by the installer and run
- The init is minimal: 206 lines
- So it really is essentially all source

“U” is for “Universal”

- A single flash/ssd/usb device can boot on all Go-supported architectures
- Adding a new architecture requires only 4 new binaries
- 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
  - or wants to wait for compilation
- Some users want to eliminate fork/exec
- Some FLASH parts are too small for all of the source
- Hence the root image can take many forms
- But source code never changes
  - I.e. no specialized source code for embedded

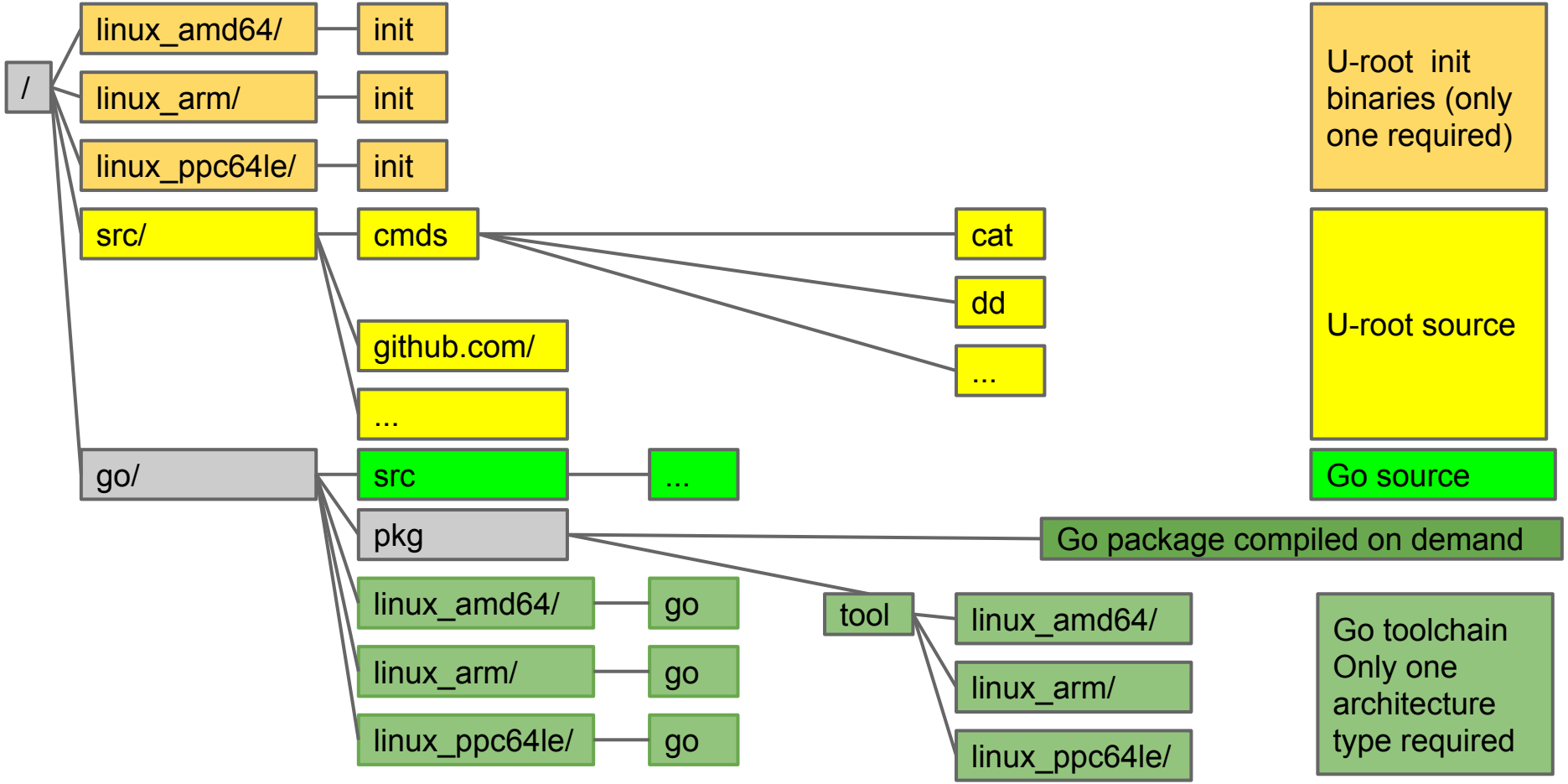
Variations of u-root

5 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 5 binaries per architecture: some/all commands precompiled, dynamic compilation, multiple architectures in one root image	Post-boot model where faster boot is required	
5 binaries, all commands in source form, dynamic compilation, one architecture	Pre-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	
One binary, but no fork/exec	Almost always firmware	MIN

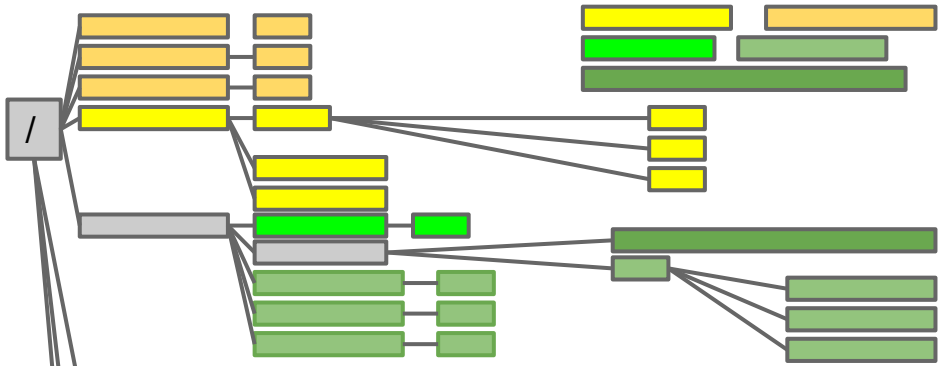
A deeper look at u-root “MAX”

- Standard kernel
- five/four/two Go binaries *per architecture**
  - init binary (part of u-root, written in Go)
  - “go” command, xg, [xc],xa, xl (e.g. 6, 5, 8, q)
- Go package **source**
- u-root source for basic commands
  - (cat, dd, etc.)
- in 13M (compressed of course! :-)

Root structure at boot



Init builds directories, mounts, ...



buildbin/

sh -> installcommand
cat -> installcommand
...

installcommand

bin/

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

Directory of symlinks built by init

installer binary

Init creates required device nodes, mount points, and mounts

Scripting and Builtins are essential

- u-root shell can take simple scripts
  - i.e. files with basic commands, pipelines, etc.
- Doesn’t support (e.g.) functions
- We wanted a simple shell
  - Avoid complexity and bugs of traditional shell scripting environments
- Want one language for tools and scripts
- Make Go our scripting language

Using Go to write more Go

- For scripting
- For dynamically creating shells with builtins
- For creating small memory pre-compiled versions

Script for ip link command

```
script { 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?

Go script rewrites fragment and uses the go import package

- script 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

- Take a fragment and produce buildable Go code
- Script program builds and runs the code
- Builds on existing u-root mechanisms but uses Go to write new Go

```
package main
import “net”
import “fmt”
func main(){
...
}
```

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 script
- 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

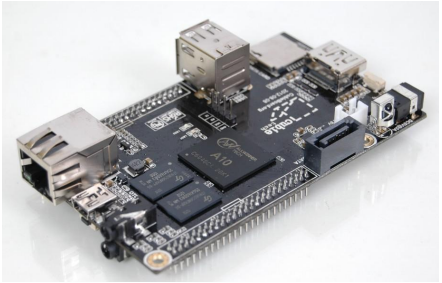
Customize the shell in a few steps

- 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 /bin
- mount --bind the tempdir over /src/cmds/sh/
- run /buildbin/sh
- 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 shell finally exits, the 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

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 and avoid fork/exec if possible

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 main() {
}
```

```
package ls

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

- Shell does not call exec; ls becomes builtin
- Combine all of u-root into one program

u-root as one program

- No need for fork/exec
- The Go size penalty is gone
  - one program: about 2M
  - 33 programs: about 7M
- compresses to 400K
- And we get dhcp, wget, ping, ip, …
- Processing 33 commands takes .2 seconds
- Full build takes 12 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

Where to get it

- Prebuilt docker with the whole system
  - docker.io rminnich/u-root (use tag 18 or “latest”)
  - sudo /usr/local/bin//docker run --privileged -i -t rminnich/u-root:18 /bin/bash
  - cd /u-root && sh README to try chroot
  - /coreboot, /go, /u-root, and /linux-3.14.17 so you can build it all from source to try it yourself
- github.com/rminnich/u-root

Extra

Init tasks

- /bin 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:/buildbin
- create /dev, /proc, /etc
- Create inodes in /dev
- mount procfs
- Create minimal /etc/resolv.conv

Running first sh

- Init then execs sh
- If sh is not in /bin, /buildbin/sh (symlink->installcommand) runs
- /buildbin/installcommand directs go to build sh, and then execs /bin/sh
- And you have a shell prompt
- Same flow for other programs

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!)

But I want bash!

- You want it, you got it: tinycore 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