u-root: / with the convenience of scripting and the performance of compilation

Ron Minnich
Google

Andrey Mirtchovski
Cisco

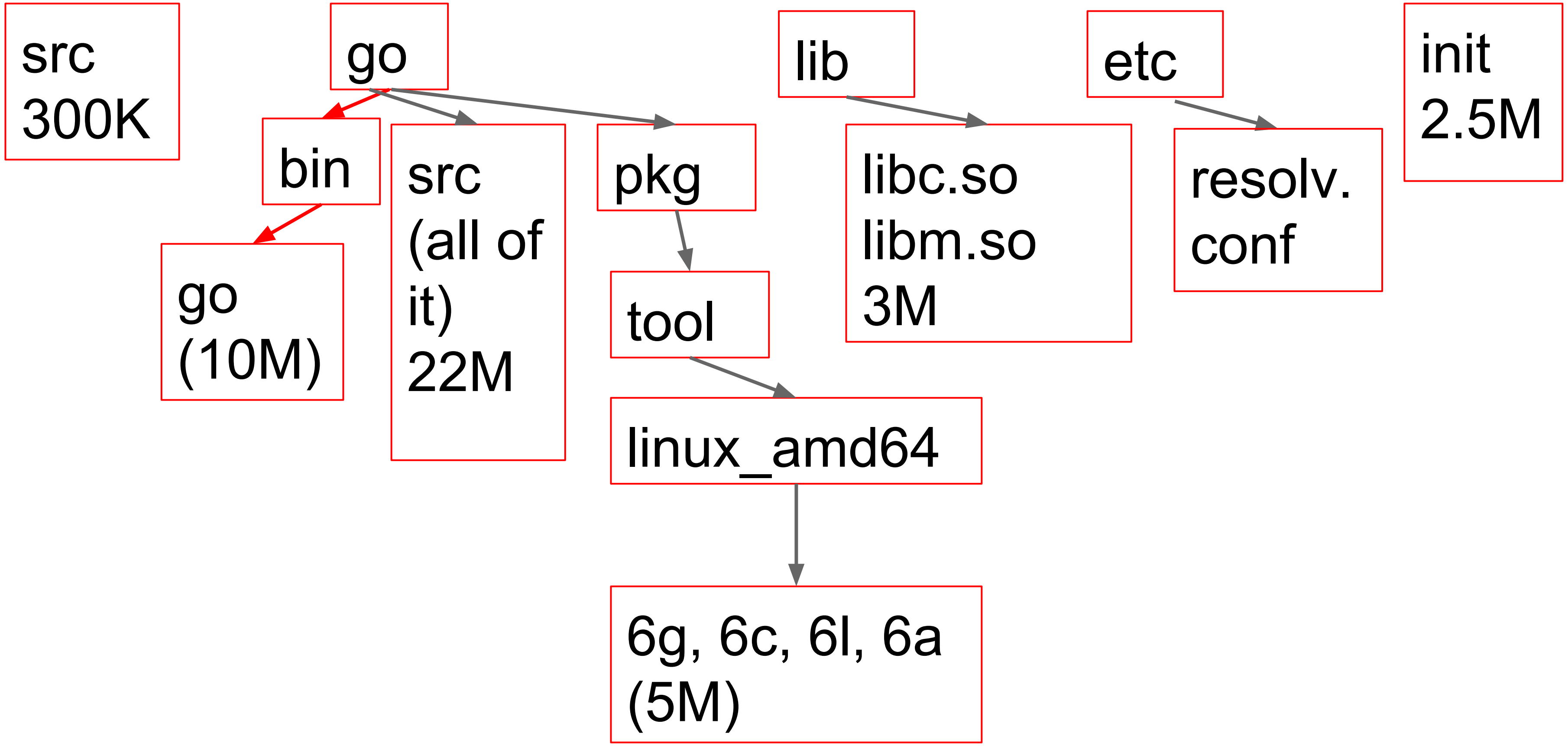
Outline

- What u-root is
- Why we’re doing it
- How it all works
- Try it!
  - sudo docker -g [in its own window]
  - sudo docker run --privileged -i -t rminnich/u-root:18 /bin/bash
  - cd /u-root && sh README

U-root: a small source-based root

- Standard kernel
- five Go binaries: go, 6c, 6g, 6l, 6a
- Go package **source** (aka libraries)
- Set of u-root source files for basic commands (cat, dd, etc.)
- One binary for u-root: init
- in 10M (compressed :-)
- i.e., a source-based root file system

Root directory structure



Basic Model

- /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
- When, e.g., cat is executed, if not in /bin, /buildbin/cat (symlink->installcommand) runs
- /buildbin/installcommand directs go to build the command, and then execs it

A Word on Go build

- Builds programs and packages
- Possible because of the ‘import’ keyword and Go package structure
- No need, therefore, to build *all* packages to build *a* program: *only* packages you need
- Contrast with glibc: (e.g.) sunrpc gets built, even if you never use it

Why build a source-based root?

- Want to embed a full busybox style environment in flash for coreboot
- The various embedded toolchain builds are large, hard to build, hard to diagnose
    - It’s hard to create a configuration you can understand later
- The busybox code can be hard to read
- C language and build system(s) the problem

The problem with C

- C is obsolete (for user mode that is)
- Too easy to write buggy, insecure code
- Lacking in modern features
- No dependency analysis possible
- Build systems built on build systems
  - configure, automake, libtool, ….
- Each trying to solve each other’s problems
  - Solving complexity with complexity

But C can produce compact code!

- For building a tight, small binary, busybox continues to impress
- Can get a useful set of binaries in about 1M
  - Single Go binary: 1MiB
- But BIOS FLASH is 16-32M!(thank u EFI!)
- If we have 32MiB can we change the way we deliver binaries?
- Could we just ship source?

Summary: Why u-root?

- Want “busybox” in 16 or 32 MiB
- Want a modern compiled language
  - Which, unfortunately, means big binaries
- Source in a modern language is much *smaller* than equivalent source in C
- Conclusion: ship source, it’s small
- Build programs, libraries on-demand
  - Implies compilation must be *fast, package-oriented*

Enter Go

- Type safe
- Garbage collected
- Dependency analysis from source
- Very fast compilation
    - Most things compile in a second or so
- Expressive, hence small source
- Can ship source, compile on demand

Go packs a lot in a small space

- E.G.: show all net interfaces and addresses:

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

```
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]
```

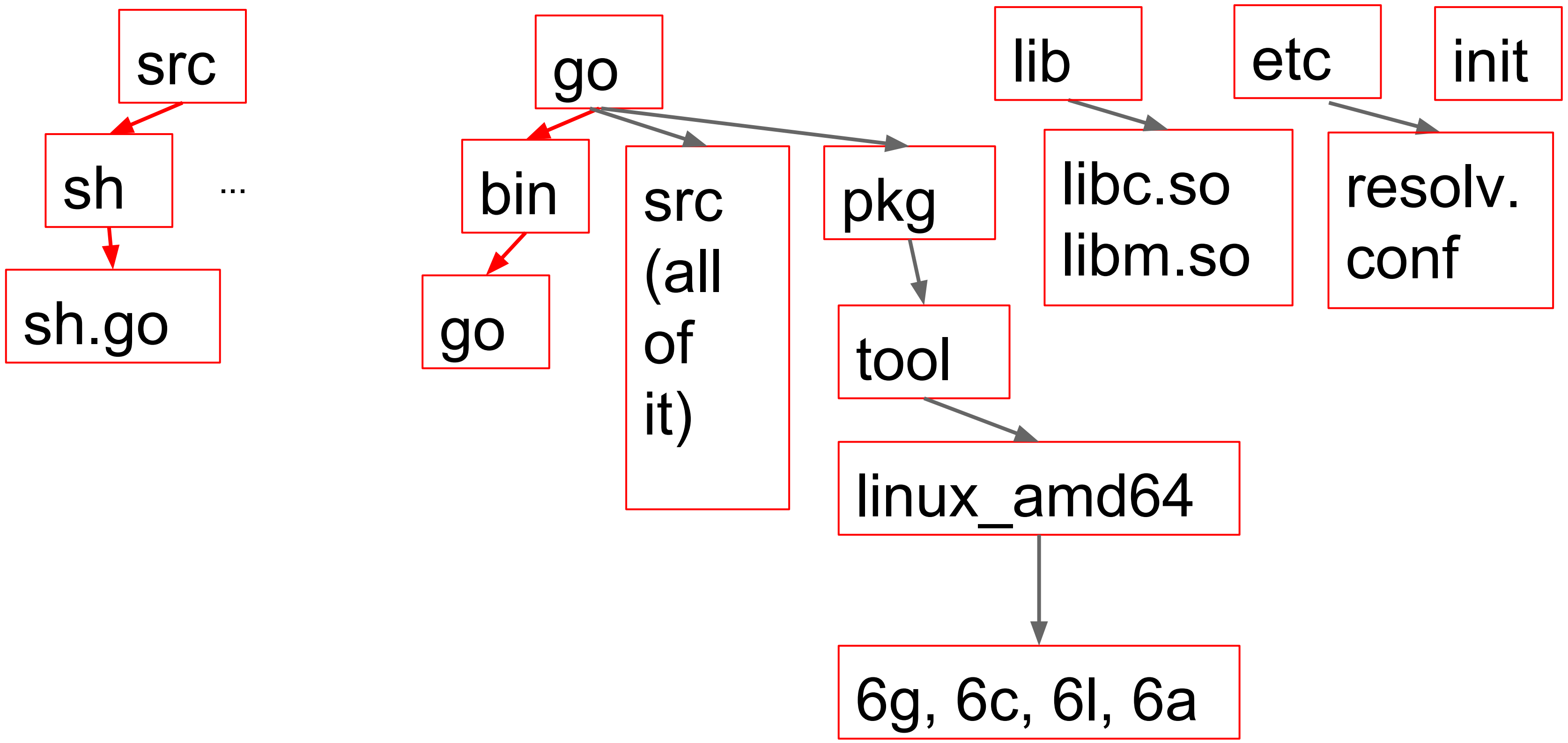
How u-root works

- u-root file system consists of:
  - toolchain to build go programs
  - source for packages needed for these programs
  - (still small) set of standard commands
  - One binary (init) to start things up
- Four ways to try it out
  - chroot
  - Boot kernel in qemu (i.e. --kernel)
  - Boot from disk via e.g. syslinux
  - Boot from coreboot qemu

Setting up u-root file system

- Initially have src/ (i.e. src/sh/sh.go, …)
- Script finds all packages needed, creates
  - go/bin/go (build tool)
  - go/src/pkg/…
    - coreboot case: only required packages
  - go/pkg/.../6[a,l,c,g] (compilers)
- Pull in two .so’s for compilers, resolv.conf
- builds src/init/init.go, places it in init

Setting up Root directory structure



Commands

What we have

- sh, bin, date, cat, …
- ip (set up network)
- ping, netcat
- mount
- simple web server

What we need

- insmod, …
- dhcp
- WIFI tools
- Full mount for all file systems

At boot, /bin/init starts

/src/... /go/... init

pre-built root from previous slide

bin

buildbin

installcommand
symlinks: for all src/
e.g.
cat->installcommand
dd->installcommand
sh->installcommand

/dev,
/proc,
...

Last step: run /buildbin/sh

When user types /bin/date

- /bin/date not found, but /buildbin/date is
    - symlink to installcommand
- installcommand starts, grabs arg[0], runs go build with that arg
- go build figures out what pkgs are needed, builds them, then builds date into /bin/date
- installcommand then execs /bin/date
- 237 ms for date; 2 seconds with packages

HP FALCO 2-core chromebook, 4GiB

- First build of all packages for /bin/installcommand ~5s
  - about 162 commands, many more files
- Subsequent commands are much faster because more packages are already built
- Date + 2 packages is 1 second
- 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

Status

- Work in progress
- But offers:
  - a better language than C
  - easier (much!) build path for binutils
  - Ability to use existing tools via tinycore
  - Want to know how something works? The source is right there
- Could we write systemd/uselessd in Go?

Extending the shell

```
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

Interesting futures

- Can put go binaries for ARM in the root
- Could create a usb stick that boots on ARM and x86
- And most of it is source-based
- You can pre-build some things into /bin when you build the root image
- and later remove them, to get a new version

What is this good for?

- Knowing how things work
- Much easier embedded root than busybox
- Security that comes from source-based root
- 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

Considerations

- First instance of execution can be long (since you’re building it)
- If u-root doesn’t have “it” you need a working network to get tinycore packages
- Lots of things missing (e.g. WIFI, bluetooth support)
    - Come and help out! You’ll learn a lot!

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