U-root: A Go-based, Firmware Embeddable Root File System with On-demand Compilation

Ronald G. Minnich, *Google;* Andrey Mirtchovski, *Cisco*

https://www.usenix.org/conference/atc15/technical-session/presentation/minnich

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).
July 8–10, 2015 • Santa Clara, CA, USA
ISBN 978-1-931971-225

U-root: A Go-based, firmware embeddable root file system with on-demand compilation

Ronald G. Minnich
Google

Andrey Mirtchovski
Cisco

Abstract

U-root is an embeddable root file system intended to be placed in a FLASH device as part of the firmware image, along with a Linux kernel. The program source code is installed in the root file system contained in the firmware FLASH part and compiled on demand. All the u-root utilities, roughly corresponding to standard Unix utilities, are written in Go, a modern, type-safe language with garbage collection and language-level support for concurrency and inter-process communication.

Unlike most embedded root file systems, which consist largely of binaries, U-root has only five: an init program and 4 Go compiler binaries. When a program is first run, it and any not-yet-built packages it uses are compiled to a RAM-based file system. The first invocation of a program takes a fraction of a second, as it is compiled. Packages are only compiled once, so the slowest build is always the first one, on boot, which takes about 3 seconds. Subsequent invocations are very fast, usually a millisecond or so.

U-root blurs the line between script-based distros such as Perl Linux[24] and binary-based distros such as BusyBox[26]; it has the flexibility of Perl Linux and the performance of BusyBox. Scripts and builtins are written in Go, not a shell scripting language. U-root is a new way to package and distribute file systems for embedded systems, and the use of Go promises a dramatic improvement in their security.

Introduction

Embedding kernels and root file systems in BIOS FLASH is a common technique for gaining boot time performance and platform customization[25][14][23]. Almost all new firmware includes a multiprocess operating system with a full complement of file systems, network drivers, and protocol stacks, contained in an embedded file system. In some cases, the kernel is only booted long enough to boot another kernel; in others, the kernel that is booted and the file system it contains constitute the operational environment of the device[15].

These so-called "embedded root file systems" also contain a set of standard Unix-style programs used for both normal operation and maintenance. Space on the device is at a premium, so these programs are usually written in C using, e.g., the BusyBox toolkit[26]; or in an interpretive languages, such as Perl[24] or Forth. BusyBox in particular has found wide usage in embedded appliance environments, as the entire root file system can be contained in under one MiB.

Embedded systems, which were once standalone, are now almost always network connected. Network-connected systems face a far more challenging security environment than even a few years ago. In response to the many successful attacks against shell interpreters[11] and C programs[8], we have started to look at using a more secure, modern language in embedded root file systems, namely, Go[21][16].

Go is a new systems programming language created by Google. Go has strong typing; language level support for concurrency; inter-process communication via channels, a la Occam[13], Limbo[17], and Alef[27]; runtime type safety and other protective measures; dynamic allocation and garbage collection; closures; and a package syntax, similar to Java, that makes it easy to determine what packages a given program needs.

The modern language constructs make Go a much safer language than C. This safety is critical for network-attached embedded systems, which usually have network utilities written in C, including web servers, network servers including sshd, and programs that provide access to a command interpreter, itself written in C. All are proving to be vulnerable to the attack-rich environment that the Internet has become. Buffer overflow attacks affecting C-based firmware code (among other things) in 2015 include GHOST and the so-called *FSVariable.c* bug in Intel's UEFI firmware. Buffer overflows in Intel's

UEFI and Active Management Technology (AMT)have also been discovered in several versions in recent years. Both UEFI[12] and AMT[4] are embedded operating systems, loaded from FLASH, that run network-facing software; attacks against UEFI have been extensively studied[9]. Most printers are network-attached and are a very popular exploitation target[6].

Firmware is not visible to most users and is updated much less frequently (if at all) than programs. It is the first software to run, at power on reset. Exploits in firmware are extremely difficult to detect, because firmware is designed to be as invisible as possible. Firmware is extremely complex; UEFI is roughly equivalent in size and capability to a Unix kernel. Firmware is usually closed and proprietary, with nowhere near the level of testing of kernels. These properties make firmware an ideal place for so-called advanced persistent threats[10][18][5]. Once an exploit is installed, it is almost impossible to remove, since the exploit can inhibit its removal by corrupting the firmware update process. The only sure way to mitigate a firmware exploit is to destroy the hardware.

Even the most skilled programmers make simple mistakes that in C can be fatal, especially on network-connected systems; nowadays, even the lowest-level firmware in our PCs, printers, and thermostats is network-connected. These mistakes are either impossible to make in Go or, if made, are detected at runtime and result in the program exiting. Perhaps surprisingly, the case for using a high-level, safe language like Go in very low level embedded firmware might be stronger than for user programs, because exploits at the firmware level are nearly impossible to detect and mitigate.

The challenge to using Go in a storage-constrained environment such as firmware is that advanced language features lead to big binaries. Even a date program is about 2 MiB. One Go binary, implementing one function, is twice as large as a BusyBox binary implementing many functions. As of this writing, a typical BIOS FLASH part is 16 MiB. Fitting many Go binaries into a single BIOS flash part is not practical.

The Go compiler is very fast and its sheer speed points to a solution: to compile programs only when they are used. We can build a root file system which has almost no binaries except the Go compiler itself. The compiled programs and packages can be saved to a RAM-based file system.

U-root is our proof of concept of this idea. U-root contains only 5 binaries, 4 of them from the Go toolchain, and the 5th an init binary. The rest of the programs are contained in BIOS FLASH in source form, including packages. The search path is arranged so that when a command is invoked, if it is not in /bin, an installer is invoked instead which compiles the program into /bin; if the build succeeds, the command is executed. This first invocation takes a fraction of a second, depending on program complexity; after that, the RAM-based, statically linked binaries run in about a millisecond.

U-root blurs the boundary between script-based root file systems such as Perl Linux[24] and binary-based root file systems such as BusyBox[26]; it has the flexibility of Perl Linux and the performance of BusyBox. Scripts are written in Go, not a shell scripting language, with two benefits: the shell can be simple, with fewer corner cases; and the scripting environment is substantially improved, since Go is more powerful than most shell scripting languages, but also less fragile and less prone to parsing bugs.

The U-root design

The u-root boot image is a build toolchain and a set of programs in source form. When first used, a program and any needed but not-yet-built packages are built and installed, typically in a fraction of a second. On second and later uses, the binary is executed. The root file system is almost entirely unformed on boot; /init sets up the key directories and mounts, including common ones such as /etc and /proc.

Since the init program itself is only 132 lines of code and is easy to change, the structure is very flexible and allows for many use cases.

- Additional binaries: if the 3 seconds it takes to get to a shell is too long (some applications such as automotive computing require 800 ms startup time), and there is room in FLASH, some programs can be precompiled into /bin.

- Build it all on boot: if on-demand compilation is not desired, a background thread in the init process can build all the programs on boot.

- Selectively remove binaries after use: if RAM space is at a premium, once booted, a script can remove everything in /bin; those things that are used will be rebuilt on demand.

- Always build on demand: it is possible to run in a mode in which programs are never written to /bin and always rebuilt on demand; this mode is surprisingly comfortable to use, given that program compilation is so fast[1].

- Lockdown: if desired, the system can be locked down once booted in one of several ways: the entire /src tree can be removed, for example, or just the compiler toolchain can be deleted.

How u-root works

U-root is packaged as an LZMA-compressed initial RAM file system (initramfs) in cpio format, contained in a Linux compressed kernel image, a.k.a. bzImage. The bootloader (e.g. syslinux) or firmware (e.g. coreboot) loads the bzImage into memory and starts it. The Linux kernel sets up a RAM-based root file system and unpacks the u-root file system into it. This initial root file system contains a Go toolchain (4 binaries), an init binary, the u-root program source, and the entire Go source tree, which provides packages needed for u-root programs.

All Unix systems start an init process on boot and u-root is no exception. The init for u-root sets up some basic directories, symlinks, and files; builds a command installer; and invokes the shell. We describe this process in more detail below. The boot file system layout is shown in Table 1.

The src directory is where programs and u-root packages live. The go/bin directory is for any Go tools built after boot; the go/pkg/tool directory contains binaries for various architecture/kernel combinations. The directory in which a compiler toolchain is placed provides information about the target OS and architecture; for example, the Go build places binaries for Linux on x86_64 in in /go/pkg/tool/linux_amd64/. Note that there is no /bin or many of the other directories expected in a root file system. The init binary builds them. The u-root root file system has very little state.

For most programs to work, the file system must be more complete. We save space in the image by having init create additional file system structure at boot time: it fills in the missing parts of the root filesystem. It creates /dev and /proc and mounts them. It creates an empty /bin which is filled with binaries on demand. We show it in Table 2.

Note that in addition to /bin, there is a directory called /buildbin. Buildbin and the correct setup of $PATH are the keys to making on-demand compilation work. The init process sets $PATH to `/go/bin:/bin:/buildbin:/usr/local/bin`. Init also builds the `installcommand`, using the go bootstrap builder; and creates a complete set of symlinks as shown. As a final step, init execs sh.

There is no /bin/sh at this point; the first sh found in $PATH is /buildbin/sh. This is a symlink to installcommand. Installcommand, once started, examines argv[0], which is sh, and takes this as instruction to build /src/cmds/sh/*.go into /bin and then exec /bin/sh. There is no difference between starting the first shell and any other program.

Hence, part of the boot process involves the construction of an installation tool to build a binary for a shell which is then run. If a user wants to examine the source

Table 1: The initial layout of a u-root file system. All Go compiler and runtime source is included under /go/src; all u-root source under /src; and the compiler toolchain binaries under /go/pkg.

/src	cmds/	
		builtin/builtin.go
		cat/cat.go
		cmp/cmp.go
		comm/comm.go
		cp/cp.go
		date/date.go
		dmesg/dmesg.go
		echo/echo.go
		freq/freq.go
		grep/grep.go
		init/init.go
		installcommand/installcommand.go
		ip/ip.go
		ldd/ldd.go
		losetup/losetup.go
		ls/ls.go
		mkdir/mkdir.go
		mount/mount.go
		netcat/netcat.go
		ping/ping.go
		printenv/printenv.go
		rm/rm.go
		script/script.go
		seq/seq.go
		sh/{cd.go,parse.go,sh.go,time.go}
		srvfiles/srvfiles.go
		tcz/tcz.go
		tee/tee.go
		uniq/uniq.go
		wc/wc.go
		wget/wget.go
		which/which.go
	pkg/	dhcp/ (dhcp package source)
		netlib/ (netlib package source)
		golang.org (import package source)
/go	src/	Packages and toolchain
	pkg/	tool/linux_amd64/{6a,6c,6g,6l}
	misc/	...
	tool/	...
	bin/	go
	include/	...
/lib/	libc.so	Needed for tinycore linux packages
	libm.so	

Table 2: Layout after /init has run. /buildbin contains symlinks to enable the on-demand compilation, and other standard directories and mount points are ready.

/	Root file system from Table 1
/buildbin/ (built/installed by /init) (/init creates symlinks)	(created by /init) installcommand builtin->installcommand cat->installcommand cmp->installcommand comm->installcommand cp->installcommand date->installcommand dhcp->installcommand dmesg->installcommand echo->installcommand freq->installcommand grep->installcommand init->installcommand ip->installcommand ldd->installcommand losetup->installcommand ls->installcommand mkdir->installcommand mount->installcommand netcat->installcommand ping->installcommand printenv->installcommand rm->installcommand script->installcommand seq->installcommand sh->installcommand srvfiles->installcommand tcz->installcommand tee->installcommand uniq->installcommand wc->installcommand wget->installcommand which->installcommand
/bin	/init creates
/proc	/init mounts /proc
/tcz	/init creates for tinycore binaries
/dev	init creates minimal needed devices
/etc	init writes resolv.conf

to the shell, they can cat /src/cmds/sh/*.go; the cat command will be built and then show those files.

U-root is intended for network-based devices, and hence good network initialization code is essential. U-root includes a Go version of the `ip` and `dhcp` programs, along with the docker `netlink` package and a dhcp package. Support for WIFI configuration is underway.

The u-root shell

A shell is a key part of any boot system. Shells run commands, where a command is a sequence of one or more programs, potentially tied together with pipes or other operators. Shells may run scripts from a file. Scripts are usually simple sequences of commands, each command invoking just one program, but the shell language may allow more complex commands in a script. Shells have built-in commands, i.e. commands that do not invoke a program, but are recognized by the shell and executed directly. Builtins are used when the command must change the shell state, as in the `cd` command; when the cost of starting a program is felt to be too high relative to the operation the command performs; because the shell source is not available or it is too hard to change the shell; or for convenience, i.e. users would rather write in the shell language instead of C. Most shells can be extended via a builtin facility, which usually looks like a function definition style syntax. The shell scripting language is usually the same language used for builtins.

Every boot loader in common use today has some sort of shell capability. That these shells have many limitations is a given, but at the same time they need to look as much as possible like a standard shell.

U-root provides a shell that is stripped down to the fundamentals: it can read commands in, using the Go scanner package; it can expand (i.e. glob) the command elements, using the Go filepath package; and it can run the resulting commands, either programs or shell builtins. It supports pipelines and IO redirection. At the same time, the shell defines no language of its own for scripting and builtins; instead, the u-root shell uses the Go compiler. In that sense, the u-root shell reflects a break in important ways with the last few decades of shell development, which has seen shells and their language grow ever more complex and, partially as a result, ever more insecure[19] and fragile[11].

The shell has several builtin commands, and the user can extend it with builtin commands of their own. Before we discuss user-defined builtins, we will describe the basic source structure of u-root shell builtins.

All shell builtins, including the ones that come with the shell by default, are written with a standard Go init pattern which installs one or more builtins. Shown in

Figure 1 and 2 is the shell builtin for time.

Builtins in the shell are defined by a name and a function. One or more builtins can be described in a source file. The name is kept in a map and the map is searched for a command name before looking in the file system. The function must accept a string as a name and a (possibly zero-length) array of string arguments, and return an error. In order to connect the builtin to the map, the programmer must provide an init function which adds the name and function to the map. The init function is special in that it is run by Go when the program starts up. In this case, the init function just installs a builtin for the time command.

Scripting and builtins

To support scripting and builtins, u-root provides two programs: `script` and `builtin`. The `script` program allows users to specify a Go fragment on the command line, and runs that fragment as a program. The `builtin` program allows a Go fragment to be built into the shell as a new command. Builtins are persistent; the builtin command instantiates a new shell with the new command built in. Scripts run via the script command are ephemeral.

We show a usage of the script command in Figure 3.

This script implements printenv. Note that it is not a complete Go program in that it lacks a package statement, imports, a main function declaration, and a return at the end. All the boilerplate is added by the script command, which uses the Go imports package to scan the code and create the import statements required for compilation (in this case, both fmt and os packages are imported). Because our shell is so simple, there is no need to escape many of these special characters. We have offloaded the complex parsing tasks to Go.

Builtins are implemented in almost the same way. The builtin command takes the Go fragment and creates a standard shell builtin Go source file which conforms to the builtin pattern. This structure is easy to generate programmatically, building on the techniques used for the script command.

A basic hello builtin can be defined on the command line:

```
builtin hello \
'{ fmt.Printf("Hello\n") }'
```

The fragment is defined by the {} pair. Given a fragment that starts with a {, the builtin command generates all the wrapper boiler plate needed. The builtin command is slightly different from the script command in that the Go fragment is bundled into one argument. The command accepts multiple pairs of command name and Go code

```
// Package main is the 'root' of the
// package hierarchy for a program.
// This code is part of the main
// program, not another package,
// and is declared as package main.
package main

// A Go source file lists
// all the packages on which
// it has a direct dependency.
import (
  "fmt"
  "os"
  "time"
)

// init() is an optional function.
//If init() is present in a file,
// the Go compiler and runtime
// arrange for it to be called
// at program startup.
// It is hence like a constructor.
func init() {
        // addBuiltIn is provided by
        // the u-root shell for
        // the addition of builtin
        // commands. Builtins must
        // have  a standard type:
        // o The first parameter is
        //     a string
        // o The second is a string
        //     array which may be 0
        //    length
        // o The return is the Go
        //     error type
        // In this case,
        // we are creating a builtin
        // called time which calls
        // the timecmd function.
  addBuiltIn("time", timecmd)
}
```

Figure 1: The code for time builtin, Part I: setup

```
// The timecmd function is passed
// the name of a command to run,
// optional arguments,
// and returns an error. It:
// o gets the start time using Now
//   from the time package
// o runs the command using the
//     u-root shell runit function
// o computes a duration using
//     Since from the time package
// o if there is an error,
//     prints the error to os.Stderr
// o uses fmt.Printf to print
//     the duration to os.Stderr
// Note that since runtime always
// handles the error, by printing
// it, it always returns nil.
// Most builtins return the error.
// Here you can see the usage
// of the imported packages
// from the imports statement above.
func timecmd(name string, args []
    string) error {
  start := time.Now()
  err := runit(name, args)
        if err != nil {
    fmt.Fprintf(os.Stderr, "%v\n",
        err)
  }
  cost := time.Since(start)
  fmt.Printf(os.Stderr, "%v", cost)
        // This function is special
            in that
        // it handles the error, and
            hence
        // does not return an error.
        // Most other builtins return
             the
        // error.
  return nil
}
```

Figure 2: The code for the shell time builtin, Part II.

```
script \
{ fmt.Printf("%v\n", os.Environ()) }
```

Figure 3: Go fragment for a printenv script. Code structure is inserted and packages are determined automatically.

fragments, allowing multiple new builtin commands to be installed in the shell.

Builtin creates a new shell at `/bin/sh` with the source at `/src/cmds/sh/`. Invocations of `/bin/sh` by this shell and its children will use the new shell. Processes spawned by this new shell can access the new shell source and can run the builtin command again and create a shell that further extends the new shell. Processes outside the new shell's process hierarchy can not use this new shell or the builtin source. When the new shell exits, the builtins are no longer visible in any part of the file system. We use Linux mount name spaces to create this effect[22]. Once the builtin command has verified that the Go fragment is valid, it builds a new, private namespace with the shell source, including the new builtin source. From that point on, the new shell and its children will only use the new shell. The parent process and other processes outside the private namespace continue to use the old shell.

Environment variables

The u-root shell supports environment variables, but manages them differently than most Unix environments. The variables are maintained in a directory called /env; the file name corresponds to the environment variable name, and the files contents are the value. When it is starting a new process, the shell populates child process environment variables from the /env directory. The syntax is the same; $ followed by a name directs the shell to substitute the value of the variable in the argument by prepending /env to the path and reading the file.

The shell variables described above are relative paths; /env is prepended to them. In the u-root shell, the name can also be an absolute path. For example, the command `script $/home/rminnich/scripts/hello` will substitute the value of the hello script into the command line and then run the script command. The ability to place arbitrary text from a file into an argument is proving to be extremely convenient, especially for script and builtin commands.

Using external packages and programs

No root file system can provide all the packages all users want, and u-root is no exception. We must have the ability to load external packages from popular Linux distros. As a proof of concept, we created a tool to load external packages from the TinyCore Linux distribution, a.k.a. tinycore. A tinycore package is a mountable file system image, containing all the package files, including a file listing any additional package dependencies.

To load these packages, u-root provides the tcz command which fetches the package and and needed dependencies. Hence, if a user wants emacs, they need merely type tcz emacs, and emacs will become available in /usr/local/bin. The tinycore packages directory can be a persistent directory or it can be empty on each boot.

The tcz command is quite flexible as to what packages it loads and where they are loaded from. Users may specify the host name which provides the packages; the TCP port on which to connect; the version of tinycore to use; and the architecture. The tcz command must loopback mount each package as it is fetched, and hence must cache them locally. It will not refetch already cached packages. This cache can be volatile or maintained on more permanent storage. Performance varies depending on the network being used and the number of packages being loaded, but seems to average about 1 second per package on a WIFI-attached laptop.

U-root also provides a small web server, called srvfiles, that can be used to serve locally cached tinycore packages for testing. The entire server is 18 lines of Go.

Using u-root: current targets

There are three current targets for u-root. All three are available in a Docker image we provide.

The first two targets are used to test u-root to make sure it will work before it is loaded into its real target, a firmware image.

Chroot test

The first test target is a chroot environment. A chroot is a file system tree which must have at least one binary. A standard Unix command, chroot, uses the chroot system call to set the root for a child process and then execs a named binary from the tree. Note that the chroot only applies to the program being run, and does not affect any other programs.

A script provided with u-root builds an image of the file system shown in Table 1, including locating and installing the Go source tree and toolchain. The script also builds the init binary which the kernel runs as the first user-mode process.

The chroot environment simulates a full boot environment. The chroot startup process, running on a linux instance in VmWare Fusion on a Mac laptop takes about 3 seconds, including compiling the two binaries (installcommand and sh) and the packages they need, about 250 files. Once the startup process is done the user sees the u-root shell prompt and can run tests.

Kernel image

Once the file system tree has been verified via the chroot, it can be used as the input to the process of creating a so-called initramfs. An initramfs is a file system image that is built directly into a bootable Linux kernel image. When the kernel starts, it locates the initramfs, sets up a RAM file system, and extracts the intramfs into the RAM file system. At that point, the kernel can exec /init.

U-root includes the script to create this image. The result is a file, which can then be used as input to the kernel build process. The user can then boot the kernel directly in QEMU (via qemu –kernel) or drop the bzImage into a boot disk image and test that, either via qemu or booting on real hardware.

Firmware image

The end goal of u-root is to create an embedded firmware image. Linux can not run from power on reset directly; something needs to configure the platform and then load Linux from FLASH to RAM, and for that we use coreboot. We build a kernel as shown earlier, containing an initramfs, which in turn contains u-root; this in turn is built into a firmware image.

Users can take two steps to test coreboot. The first (and optional given enough confidence) is to add the kernel bzImage as a payload to a coreboot built to run in qemu. We provide a working qemu image in a Docker container to make this easy, as well as a script to add the bzImage to the coreboot image. Once the image is built, users start qemu with this image as the bios: qemu –bios coreboot.rom

Once the Linux image is known good, the user can embed it in a real coreboot image for a real mainboard. A current known limitation is that the board must contain a 16 MiB FLASH part. We have tested on the Asrock E350M1. In that case, we first tested on QEMU, then took the Linux image unchanged, merged it into the coreboot for the hardware, and it worked with no changes, indicating that QEMU provides a very accurate verification environment for the hardware target. If a given u-root build works in QEMU it will almost certainly work on hardware.

Discussion

Building the image

Building is a straightforward process, which requires a kernel source tree, Go source tree, the u-root source, and coreboot source. Build times vary depending on what infrastructure is used. In general, the kernel and Go build steps are measured in minutes, and the u-root and coreboot build times are measured in seconds. These orders of magnitude have changed little in the last 5 years.

Usability

We have been using u-root for a few months. The system provides a very usable firmware command line environment, comparable to what we have used with U-boot, UEFI, and Open Firmware. The u-root and its shell provide familiar tools and capabilities. The ability to start background tasks, shell pipelines, and redirect output to files is both useful and unusual in an embedded environment. In terms of scripting, it is more sophisticated than any shell we know of, given that our scripting language is Go.

The script command is more powerful than we first realized. It is possible to run it under any shell, or from any program. It can use much more powerful Go fragments than we have shown here, including whole Go programs, since its internal parsing is just the standard Go "imports" library. The script command fills in the blanks as needed, but only as needed.

The builtin command is perhaps the most powerful tool in u-root. It allows users to build fully customized shells for a specific purpose and then just as quickly discard them. The new shell is ready and running in less than a second. We could apply this technique to the problem of startup scripts for embedded environments. To support standard functions in init scripts, distributions provide several hundred files, and for each invocation of the shell, some set of these files are included over and over again. The time it takes to read, parse and run these standard scripts is a large part startup time[1].

With the scripting and builtin tools, users do not need to write full Go programs to get a new capability. We are finding that the basic set of tools we have is enough, and we are writing new tools as Go fragments.

Once the network is up, tools like emacs can be loaded either via a local disk, local network package server, or a remote server such as tinycorelinux.org.

Having the source always available in any sort of firmware environment is both unusual and very useful. At times, we have forgotten how some of our commands work. Having the source at hand has proven very helpful.

Future development

While we envisioned u-root as a boot time environment, we have seen increasing interest in wider use. Some users have requested common features as tab completion. The shell parser is written in such a way that it should allow for easy addition of tab completion.

History is another question, as it adds more state, complexity, and parsing to the shell. These additions in turn decrease reliability and open up paths for exploits. We are experimenting with new models of history maintenance that do not require the complexity of current systems. We might, for example, maintain history in a private per-process or per-user directory. Finding commands becomes easy, and with our extension to the shell variable model, running an old command becomes easy: $/home/rminnich/history/5 would run the fifth command. Another attraction of this model is that conversion of an old command into a shell script is easy. History stops being special to one shell and instead is common to all programs that can traverse files. Any program can see history and use the commands.

Instead of rereading the same scripts over and over, init could build a special shell at boot time that pulls in builtins to extend the shell. The scripts themselves would continue to be human-readable, but the performance of booting would be much faster, combining the perceived advantages of upstart-style scripting with systemd-level performance.

[1]A good discussion can be found in http://free-electrons.com/doc/training/boot-time/boot-time-slides.pdf

Related work

There are two main components to this work: on-demand compilaton and embedding a kernel and root file system in FLASH. Both ideas have been used at different times. Our work combines the two so that we can use Go. We review the earlier work below.

On-Demand Compilation

On-Demand compilation is one of the oldest ideas in computer science.

Slimline Open Firmware (SLOF)[7] is a FORTH-based implementation of Open Firmware developed by IBM for some of its Power and Cell processors. SLOF is capable of storing all of Open Firmware as source in the FLASH memory and compiling components to indirect threading on demand[2].

In the last few decades, as our compiler infrastructure has gotten slower and more complex, true on-demand compilation has split into two different forms. First is the on-demand compilation of source into executable byte codes, as in Python. The byte codes are not native but are more efficient than source. If the python interpreter finds the byte code it will interpret that instead of source to provide improved performance.

Java takes the process one step further with the Just In Time compilation of byte code to machine code[20] to boost performance.

Embedding kernel and root file systems in FLASH

The LinuxBIOS project[14][1], together with clustermatic[25], used an embedded kernel and simple root file system to manage supercomputing clusters. Due to space constraints of 1 MiB or less of FLASH, clusters embedded only a single-processor Linux kernel with a daemon. The daemon was a network bootloader that downloaded a more complex SMP kernel and root file system and started them. Clusters built this way were able to boot 1024 nodes in the time it took the standard PXE network boot firmware to find a working network interface.

Early versions of One Laptop Per Child used LinuxBIOS, with Linux in flash as a boot loader, to boot the eventual target. This system was very handy, as they were able to embed a full WIFI stack in flash with Linux, and could boot test OLPC images over WIFI. The continuing growth of the Linux kernel, coupled with the small FLASH size on OLPC, eventually led OLPC to move to Open Firmware.

AlphaPower shipped their Alpha nodes with a so-called Direct Boot Linux, or DBLX. This work was never published, but the code was partially released on sourceforge.net just as AlphaPower went out of business. Compaq also worked with a Linux-As-Bootloader for the iPaq.

Car computers and other embedded ARM systems frequently contain a kernel and an ext2 formatted file system in NOR FLASH, i.e. FLASH that can be treated as memory instead of a block device. Many of these kernels use the so-called eXecute In Place[3] (XIP) patch, which allows the kernel to page binaries directly from the memory-addressable FLASH rather than copying it to RAM, providing a significant savings in system startup time. A downside of this approach is that the executables can not be compressed, which puts further pressure on the need to optimize binary size. NOR FLASH is very slow, and paging from it comes at a significant performance cost. Finally, an uncompressed binary image stored in NOR FLASH has a much higher monetary cost than the same image stored in RAM since the cost per bit is so much higher.

UEFI[12] contains a non-Linux kernel (the UEFI firmware binary) and a full set of drivers, file systems, network protocol stacks, and command binaries in the firmware image. It is a full operating system environment realized as firmware.

The ONIE project[23] is a more recent realization of the Kernel-in-FLASH idea, based on Linux. ONIE packs a Linux kernel and Busybox binaries into a very small package. Since the Linux build process allows an initial RAM file system (initramfs) to be built directly into the kernel binary, some companies are now embedding ONIE images into FLASH with coreboot. Sage Engineering has shown a bzImage with a small Busybox packed into a 4M image. ONIE has brought new life to an old idea: packaging a kernel and small set of binaries in FLASH to create a fast, capable boot system.

Conclusions and future work

U-root is a root file system targeted to embedded firmware environments. In response to the increasing security challenges facing embedded systems in the always-connected Internet of Things, we have chosen to write all the u-root programs in Go, a modern, type safe language with garbage collection. The safety of the Go language and runtime reduce many of the security risks of writing network-facing services. The performance of the Go compiler makes on-demand compilation practical: most commands compile in a fraction of a second and, once compiled, run in about a millisecond.

The U-root file system, on boot, contains only 5 binaries. The rest of the root file system contains source to programs which are compiled on demand. We have found the system to be fast and usable. The images can be tested in emulation environments, of increasing fidelity to the firmware target, and have been tested on hardware running coreboot.

Our initial intent was to use u-root to build firmware images, but we are finding that we would like to use it more broadly. The structure of the Go toolchain naming scheme lends itself to heterogeneous environments: save for init, the toolchain binaries have a directory path name that includes the name of the target OS and architecture, e.g. linux_amd64, linux_arm, and so on. A single u-root image can contain many Go toolchains with no path conflicts. Were we to install more toolchains in the root file system, and move /init to the same directory containing the toolchain, a single u-root file system image could be used on many different OS and architecture combinations. We could build a u-root image, for example, that worked on all linux variants for different architectures. We are exploring this model now.

Availability

U-root is available as a git repo from

```
github.com/rminnich/u-root
```

To make trying it out easier, we have created a docker container,

```
docker.io/rminnich:18
```

The container includes all the u-root, linux kernel, and coreboot source needed to test three environments: the chroot, kernel and initramfs, and coreboot with qemu. There are scripts and logs of sessions in / which users can use to guide and verify their testing of the software.

Because u-root changes frequently, users should pull an update in /u-root once they have done initial testing.

Acknowledgements

Thanks to Maya Gokhale for her many helpful suggestions on this paper.

References

[1] AGNEW, A., SULMICKI, A., MINNICH, R., AND ARBAUGH, W. A. Flexibility in rom: A stackable open source bios. In *USENIX Annual Technical Conference, FREENIX Track* (2003), pp. 115–124.

[2] (AUTHOR OF SLOF), S. B. Personal conversation.

[3] BENAVIDES, T., TREON, J., HULBERT, J., AND CHANG, W. The enabling of an execute-in-place architecture to reduce the embedded system memory footprint and boot time. *Journal of computers 3*, 1 (2008), 79–89.

[4] BOGOWITZ, B., AND SWINFORD, T. Intel® active management technology reduces it costs with improved pc manageability. *Technology@ Intel Magazine* (2004).

[5] CELEDA, P., KREJCI, R., VYKOPAL, J., AND DRASAR, M. Embedded malware-an analysis of the chuck norris botnet. In *Computer Network Defense (EC2ND), 2010 European Conference on* (2010), IEEE, pp. 3–10.

[6] CUI, A., COSTELLO, M., AND STOLFO, S. J. When firmware modifications attack: A case study of embedded exploitation. In *NDSS* (2013).

[7] DALY, D., CHOI, J. H., MOREIRA, J. E., AND WATERLAND, A. Base operating system provisioning and bringup for a commercial supercomputer. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–7.

[8] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 475–488.

[9] KALLENBERG, C., AND BULYGIN, Y. All your boot are belong to us intel, mitre. cansecwest 2014.

[10] KALLENBERG, C., KOVAH, X., BUTTERWORTH, J., AND CORNWELL, S. Extreme privilege escalation on windows 8/uefi systems.

[11] KOZIOL, J., LITCHFIELD, D., AITEL, D., ANLEY, C., EREN, S., MEHTA, N., AND HASSELL, R. *The Shellcoder's Handbook*. Wiley Indianapolis, 2004.

[12] LEWIS, T. Uefi overview, 2007.

[13] MAY, D. Occam. *ACM Sigplan Notices 18*, 4 (1983), 69–79.

[14] MINNICH, R. G. Linuxbios at four. *Linux J. 2004*, 118 (Feb. 2004), 8–.

[15] MOON, S.-P., KIM, J.-W., BAE, K.-H., LEE, J.-C., AND SEO, D.-W. Embedded linux implementation on a commercial digital tv system. *Consumer Electronics, IEEE Transactions on 49*, 4 (Nov 2003), 1402–1407.

[16] PIKE, R. Another go at language design. Stanford University Computer Systems Laboratory Colloquium.

[17] RITCHIE, D. M. The limbo programming language. *Inferno Programmer's Manual 2* (1997).

[18] SACCO, A. L., AND ORTEGA, A. A. Persistent bios infection. In *CanSecWest Applied Security Conference* (2009).

[19] SAMPATHKUMAR, R. *Vulnerability Management for Cloud Computing-2014: A Cloud Computing Security Essential*. Rajakumar Sampathkumar, 2014.

[20] SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. Overview of the ibm java just-in-time compiler. *IBM systems Journal 39*, 1 (2000), 175–193.

[21] TEAM, G. The go programming language specification. Tech. rep., Technical Report http://golang. org/doc/doc/go spec. html, Google Inc, 2009.

[22] VAN HENSBERGEN, E., AND MINNICH, R. Grave robbers from outer space: Using 9p2000 under linux. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 83–94.

[23] VARIOUS. No papers have been published on onie; see onie.org.

[24] VARIOUS. No papers were published; see perllinux.sourceforge.net.

[25] WATSON, G. R., SOTTILE, M. J., MINNICH, R. G., CHOI, S.-E., AND HERTDRIKS, E. Pink: A 1024-node single-system image linux cluster. In *High Performance Computing and Grid in Asia Pacific Region, 2004. Proceedings. Seventh International Conference on* (2004), IEEE, pp. 454–461.

[26] WELLS, N. Busybox: A swiss army knife for linux. *Linux J. 2000*, 78es (Oct. 2000).

[27] WINTERBOTTOM, P. Alef language reference manual. *Plan 9 Programmer's Man* (1995).

Notes

[1]In fact, at one point, due to a configuration error, we were using this mode all the time without realizing it. Go compiles are that fast.