State of Intel x86 platforms today

- **Ring 3 (User)**
- **Ring 0 (Linux)**
- **Ring -1 (Xen etc.)**

Code you don’t know about:
- **Ring -2 kernel and ½ kernel**
  - Control all CPU resources.
  - Invisible to Ring -1, 0, 3
- **SMM ½ kernel. Traps to 8086 16-bit mode.**
- **UEFI kernel running in 64-bit paged mode.**

This is our focus today:
- **Ring -3 kernels**
  - Management Engine, ISH, IE.
  - Higher privilege than Ring -2.
  - Can turn on node and reimage disks invisibly. Minix 3.

X86 CPU you know about

X86 CPU(s) you don’t know about
What’s in x86 firmware?

- Mostly closed source UEFI
- Completely proprietary and potentially exploit friendly
- Controlled by vendor; hard to update without vendor support
- Varies from board to board, even on two ostentatiously identical machines
UEFI Boot

OCP Winterfell node has over 120 files in the DXE Firmware Volume
<table>
<thead>
<tr>
<th>What's in the DXE firmware volume? (and more)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CsmVideo</td>
</tr>
<tr>
<td>Terminal</td>
</tr>
<tr>
<td>SBAHCI</td>
</tr>
<tr>
<td>AHCI</td>
</tr>
<tr>
<td>AhciSmm</td>
</tr>
<tr>
<td>BIOSBLKIO</td>
</tr>
<tr>
<td>IdeSecurity</td>
</tr>
<tr>
<td>IDESMM</td>
</tr>
<tr>
<td>CSMCORE</td>
</tr>
<tr>
<td>HeciSMM</td>
</tr>
<tr>
<td>AINT13</td>
</tr>
<tr>
<td>HECIDXE</td>
</tr>
<tr>
<td>AMITSE</td>
</tr>
<tr>
<td>DpcDxe</td>
</tr>
</tbody>
</table>
LinuxBoot/NERF

Most DXEs are removed

We keep the DXE core around for ACPI and some device initialization. We remove most DXEs. We kexec next kernel.
LinuxBoot DXE FV comparison

- Only 31 files
- Most of them are SMM/SMI related DXEs and ACPI
- SMM can potentially be removed one day or at least controlled by the kernel
What’s the point?

- Control and update your firmware
- Reduce number of distinct drivers on the system
- Use Linux Kernel Engineers instead of having another UEFI team
- Remove unneeded legacy support
- Some apps/DXEs can be written as a user program in Linux
Forms of LinuxBoot

- SPI Flash
- UEFI PEI
  - Firmware HW init
  - Memory initialized
- coreboot romstage
- u-boot SPL
  - LinuxBoot
    - Linux Kernel
    - initramfs
  - Operating System
Common Questions

- Are we simply replacing GRUB?
  - No, we replace what is used to run GRUB
- Why have linux boot another linux?
  - Firmware flash size is small, you probably want a more capable runtime kernel
- Why have Go? What’s wrong with PXE?
Linux + what’s in the initramfs?

- Whatever you want.
  - We provide mechanisms, not policy.
- Stages of firmware we are replacing...
  - Drivers
  - Bootloaders
  - Debugging shells
  - ...
- Busybox?
- systemd-boot?
u-root: userspace in Go

- We have the full toolset of Linux applications at our fingertips in firmware now.
  - Let’s use them!
  - Let’s use a memory-safe language.
  - Let’s use a language that makes concurrency easy.

- u-root: 3M (compressed) initramfs in Go
  - busybox-like tools (dd, ls, cpio, ...)
  - kexec-based bootloaders (PXE- and GRUB-compatible boot tools, ...)

- LinuxBoot + u-root: NERF
- There are other runtimes: e.g. Heads.
u-root: 30 Go commands in 3M? How?!

- **Source Mode**: 6M compressed.
  - Go toolchain (compiler, linker, assembler, etc).
  - All commands in source.
  - Compiled and cached in tmpfs on the fly.
    - ~200ms to compile basic command.
  - Architecture-independent.

- **BB Mode**: 3M compressed.
  - Take all source, rewrite using AST to compile all into one binary.
  - Busybox-style: argv[0] decides what to execute.
  - Initramfs contains one binary.
Implications

- **Standard Linux shell**
  - Your firmware runs a shell you are used to!
  - No custom UEFI shells with strange commands.
  - Just use the tools you already know

- **sshd**: ssh into your firmware to debug!
  - No more bricked machines: just ssh in when it fails to boot past firmware.

- **(u-root only) init**: custom-built init in Go is faster.
  - No need for systemd, upstart, scripts.
  - Go code easier to understand than a sea of scripts
Implications (2)

- **(u-root only) Source mode**: debugging commands on the fly
  - Rewrite the source, remove the cached version, run to recompile.
  - Versatility of scripts with features and type system of Go.

- **PXE boot**
  - No more 16-bit code.
  - Trivial to use modern features.
    - HTTP(S), IPv6, ...
    - Just use a kernel & language with well-tested, audited support for them!
  - Trivial to parallelize.
    - Stop waiting for NICs to time out trying PXE boot in serial...
    - Just spawn a thread to try on each NIC.
Implications (3)

- Develop firmware applications using modern toolsets
  - Use Go static analysis tools
  - Race detector, memory sanitizer, etc...
  - Continuous Integration testing
  - Open documentation

- (Bootloader) Apps run in Ring 3 - UEFI runs them in ring 0
  - Application crashes - kernel is still up
  - ssh in and debug!
Implications (4)

- Want to write your own bootloader?
  - Hire a firmware engineer...
  - Wait, no. Just hire a normal Linux application engineer.
  - Leverage Linux knowledge already out there.

- You’re starting to get the gist...
Links

- LinuxBoot website: www.linuxboot.org
- LinuxBoot GitHub: github.com/linuxboot/linuxboot
- u-root GitHub: github.com/u-root/u-root
- Heads: www.osresearch.net