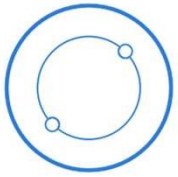
Azure Sphere: Fitting Linux Security in 4 MiB of RAM

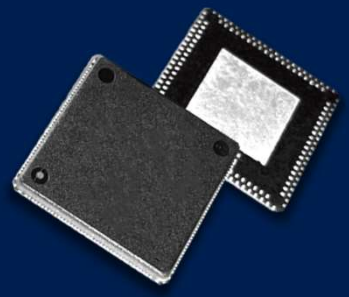
Ryan Fairfax
Principal Software Engineering Lead
Microsoft



Agenda

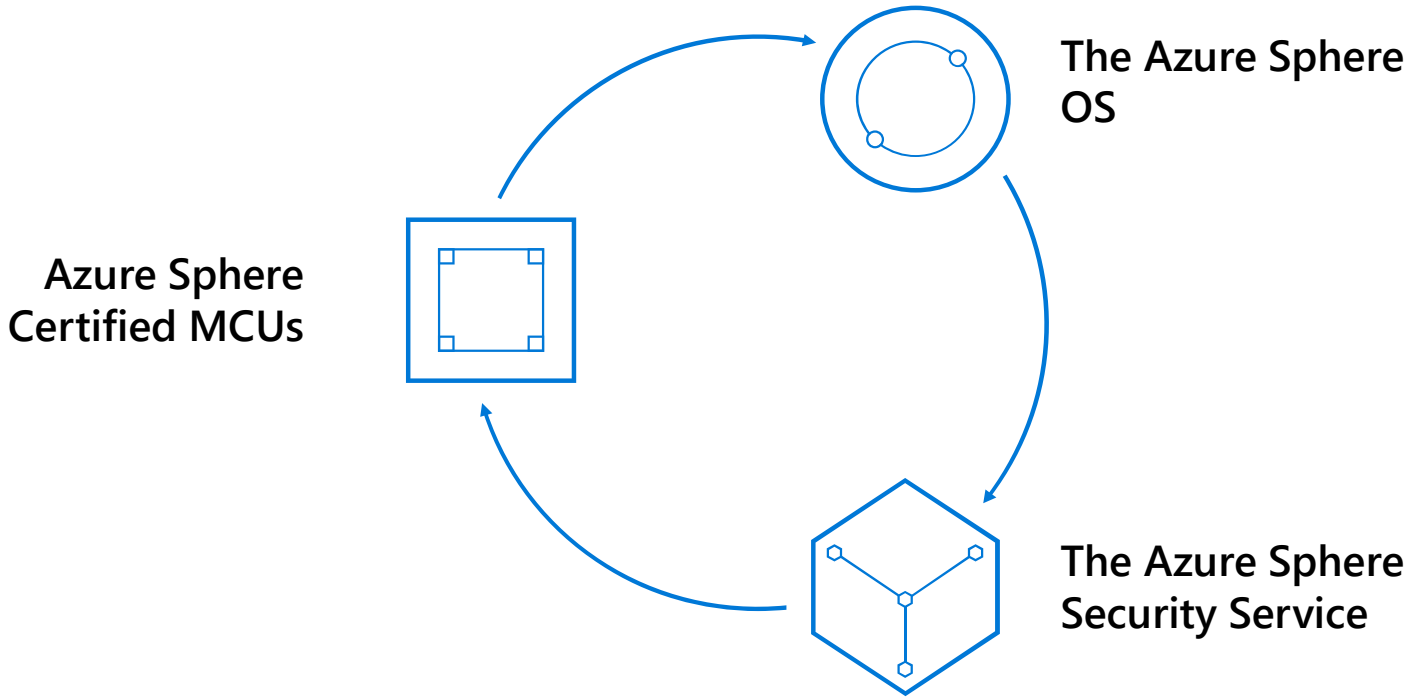
- Intro to Azure Sphere
- Kernel Customizations
- User mode services / App Model
- Future work / Takeaways

Microcontrollers (MCUs)



9 BILLION new MCU devices built and deployed every year

Azure Sphere is an end-to-end solution for MCU powered devices

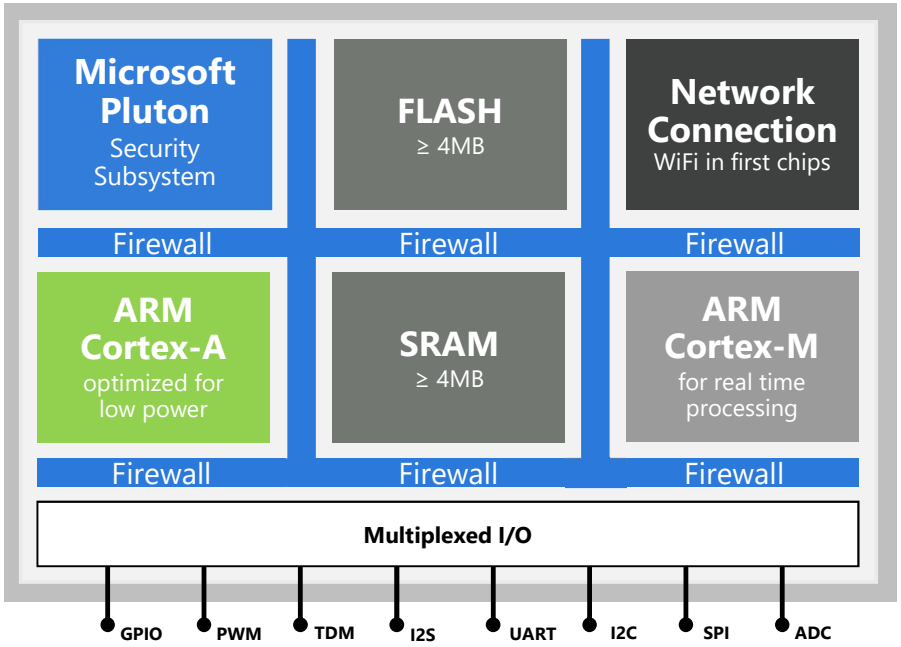


Azure Sphere certified MCUs

CONNECTED with built-in networking

SECURED with built-in Microsoft silicon security technology including the Pluton Security Subsystem

CROSSOVER Cortex-A processing power brought to MCUs for the first time



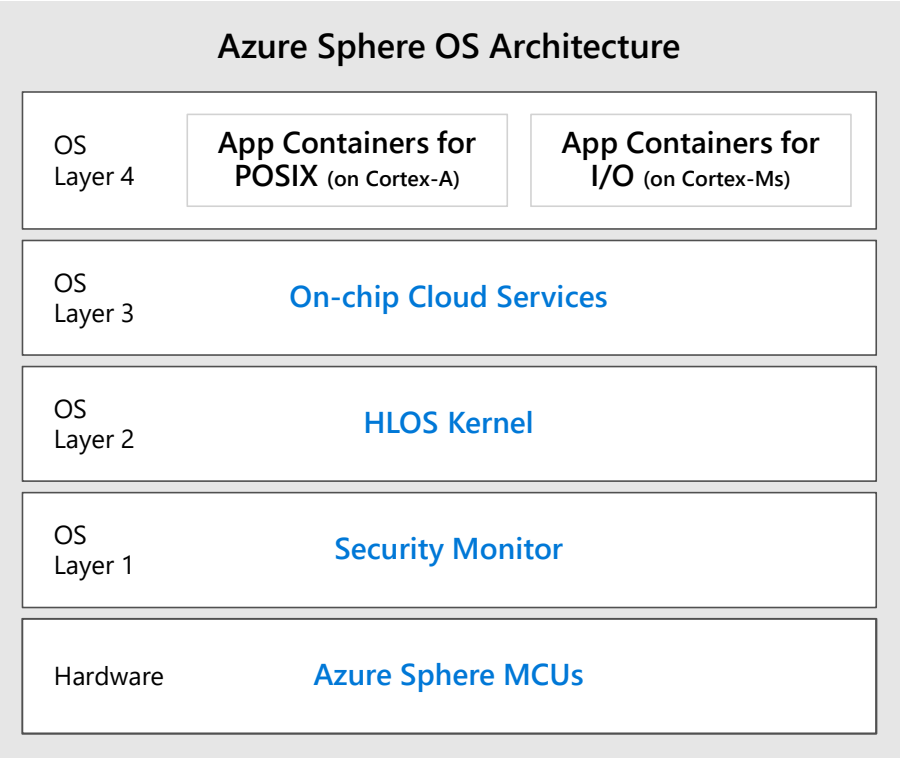
The Azure Sphere OS

Secure Application Containers
Compartmentalize code for agility, robustness & security

On-chip Cloud Services
Provide update, authentication, and connectivity

Custom Linux kernel
Empowers agile silicon evolution and reuse of code

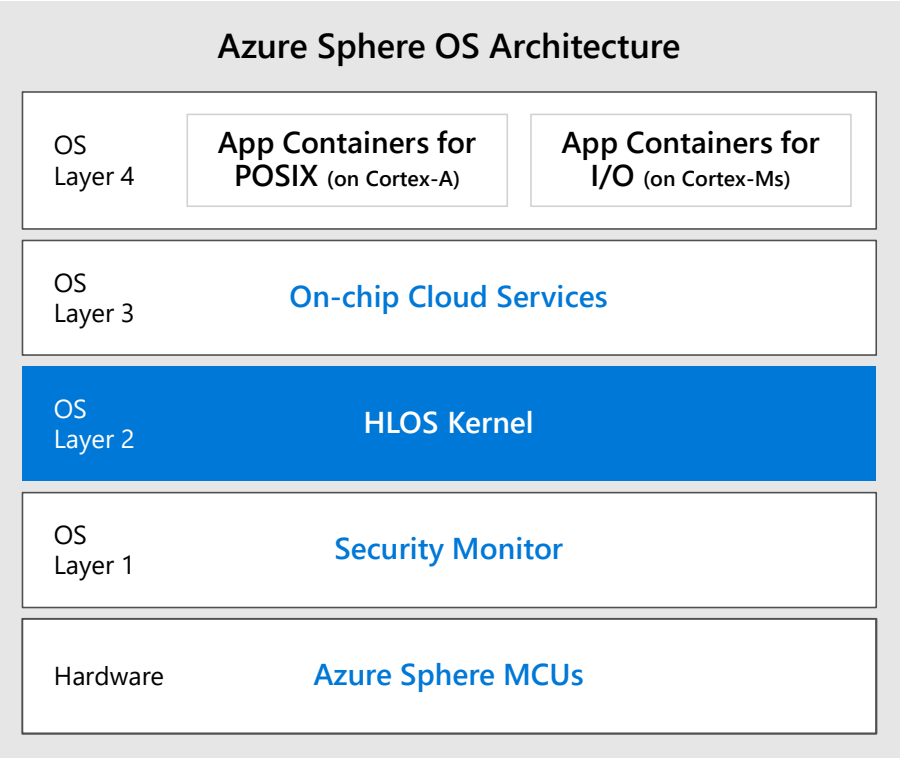
Security Monitor
Guards integrity and access to critical resources



Linux Kernel Customizations

At the heart of the Azure Sphere OS is the Linux Kernel

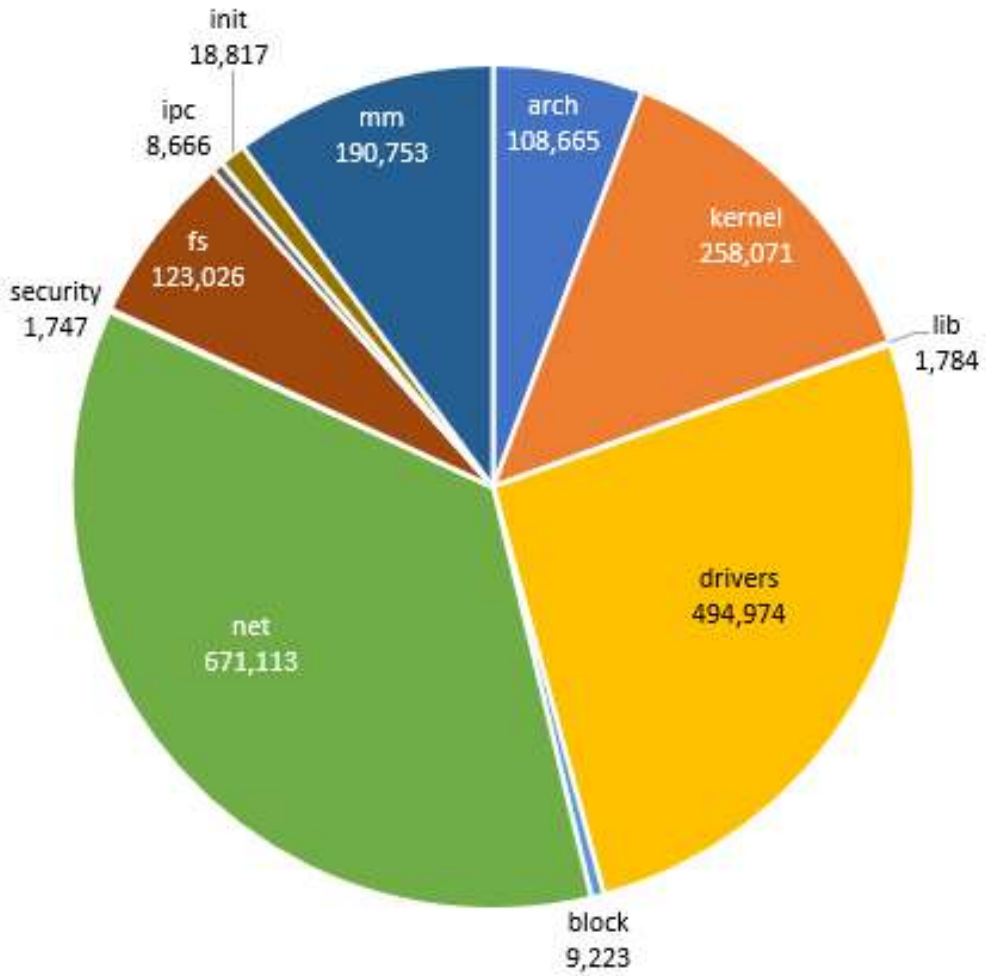
- Based on kernel.org sources
- Originally 4.1, now 4.9, with the goal to keep moving forward as LTS branches get declared
- Upstream releases are merged monthly
- We have 227 commits in a branch on top of the upstream sources as of Aug 16th



Linux Kernel Customizations

The first challenge was making it fit

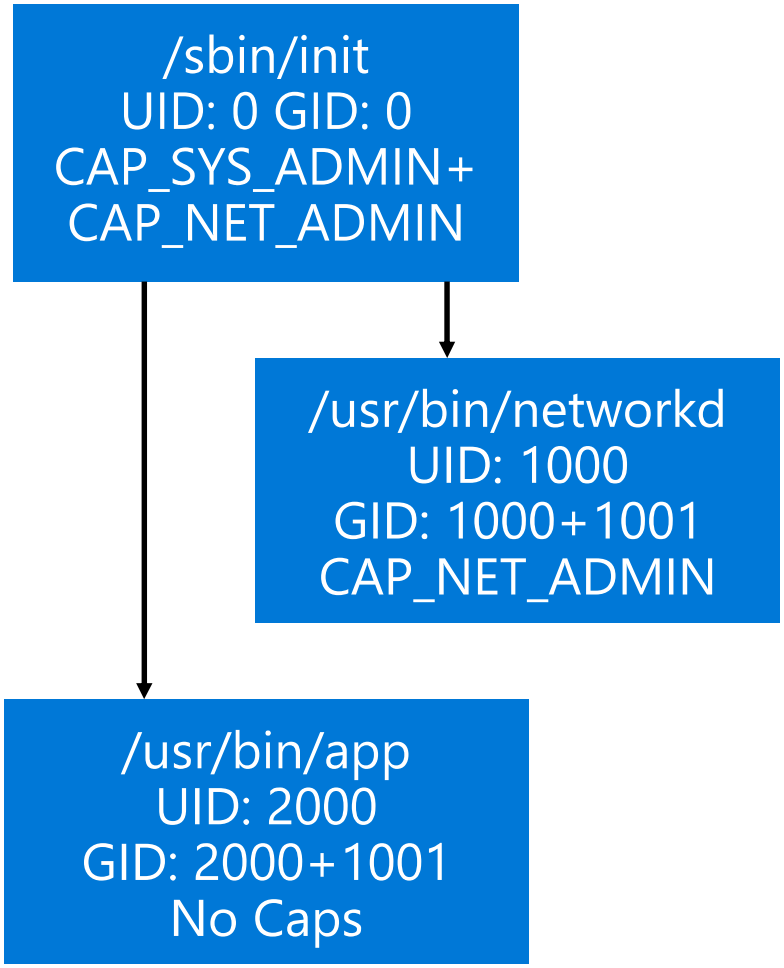
- The first build was a 4MB kernel, which took up all of the RAM
- We moved to Execute-In-Place (XIP) which helped a lot but we still used nearly 4MB of RAM to boot
- Lots of our early patches were making things more modular and tuning sizes of caches
- Some things were removed to reduce size (sysfs, most memory tracking options, kallsyms).
- As of our public preview we're at ~2.4MB of code + data, ~2100KB RAM usage after boot



Linux Kernel Customizations

The first version of our security model used static permissions baked into the filesystem

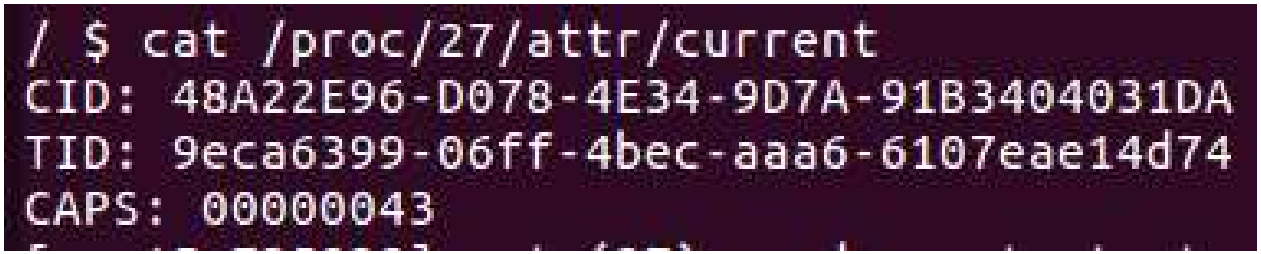
- We set owner, group, SUID, SGID on each process to ensure a consistent identity at runtime via build policy
- To pull this off we added some kernel code to enforce effective UID = real UID to avoid getting back to root
- Shared data or IPC was done via supplemental groups (also baked into filesystem)
- This made effective access for a program easier to reason about, but you had a major burden at build time to ensure the right owners / groups were set



Linux Kernel Customizations

The second version involved a simple Linux Security Module

- The goals were to reduce attack surface and enable new access control scenarios
- The LSM statically fails many operations that aren't needed on the platform
- We add three new fields to each task: App ID (CID), Network ID (TID), Capabilities



- Other apps + kernel modules can use these new fields for extended access control
- All values are immutable once set and inherit to child processes

Linux Kernel Customizations

We experimented a lot with filesystems

- We started with CRAMFS + XIP patches, but moved to a fork with modifications to reduce overhead
- We patched in CoW support for XIP code when debugging is enabled
- We tried many writable file systems that didn't work out due to RAM or flash wear overhead: ext2, jffs2, yaffs
- Ultimately we ported ARM's LittleFS to the kernel as it was a better fit for writable partitions that are hundreds of KiB in size
    - https://github.com/ARMmbed/littlefs

In some cases we added more access control to existing features

- Added per-pin GPIO access to the base GPIO infrastructure as an optional feature
- Added file system quota support for MTD devices
- Adjusted some items to leverage existing or new capabilities instead of checking for root

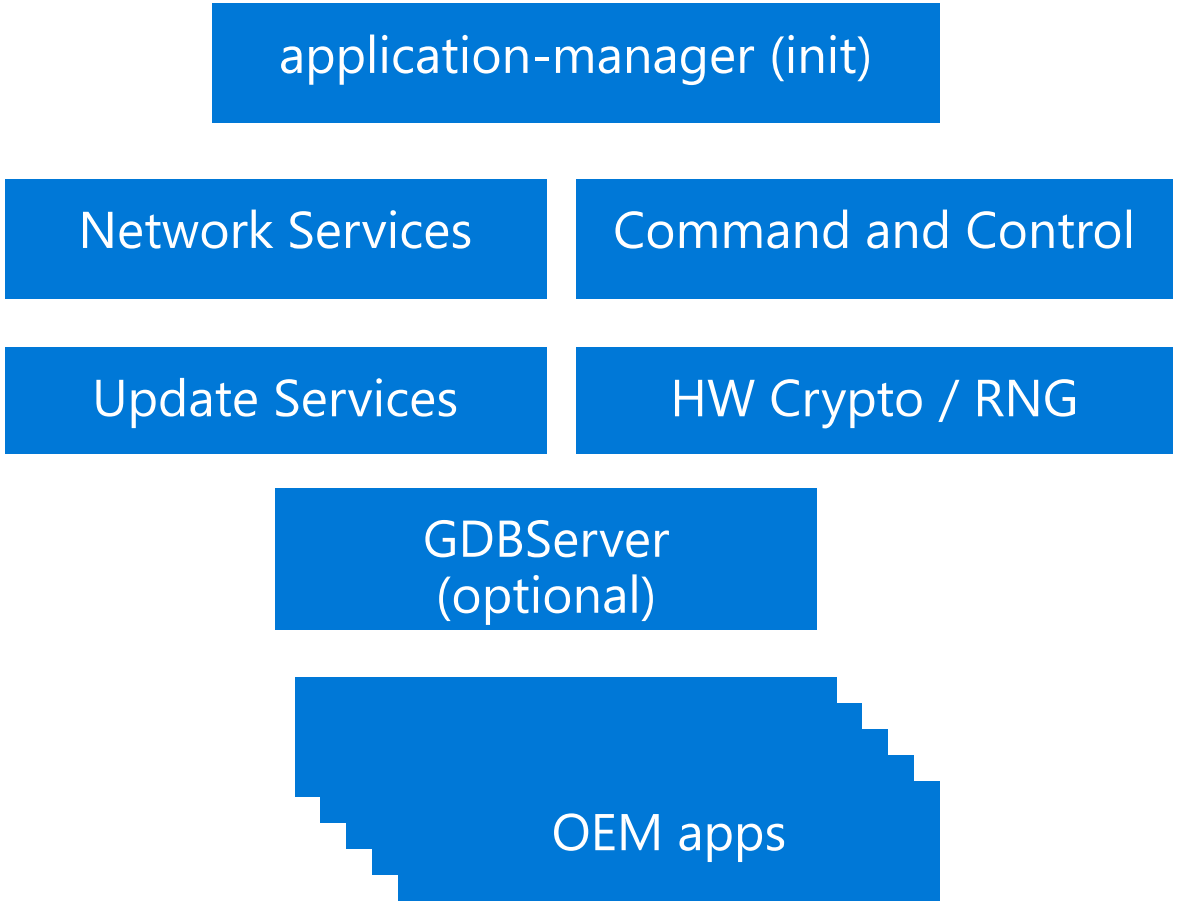
User mode / App model

Our user mode code execution model focuses around applications

- Everything is an app, other than the application manager (aka init). This includes system services.
- Apps are self describing, independently updatable, and generally run isolated from each other
- There are 4 out of box system apps + an optional debugger

Apps are renewable

- Everything on the system can be updated over the air
- Microsoft manages OS app updates, OEMs manage their app updates



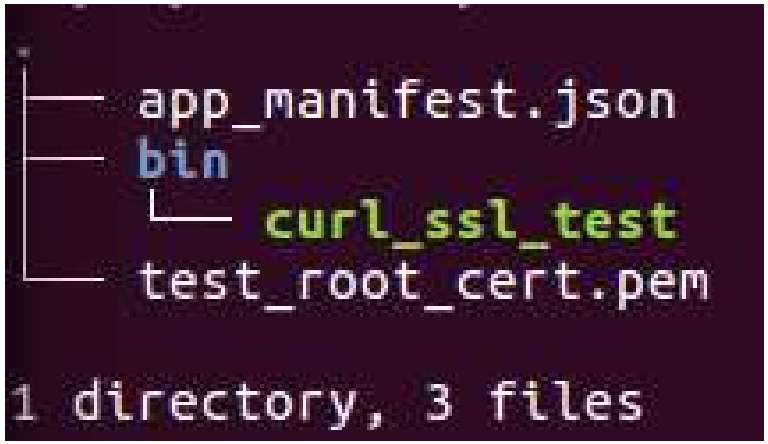
User mode / App model

We looked at options for self contained app packages and containers

- Looked seriously at LXC but couldn't get it to fit
- We started rolling our own containers with namespaces but found many of the peripherals apps used didn't interact with namespaces properly or led to capability leakage

We ended up with apps as self describing file systems

- Each app is its own filesystem that is mounted / unmounted to install / uninstall
- Apps contain metadata documents in the root of the FS that describes how they work and what they need access to
- The system validates policy and enforces it by leveraging many other Linux technologies



User mode / App model

Manifests describe the app

- What to run
- What policy to enforce
- What peripherals to allow access to

Our custom init process (application-manager) enforces the policy

- Uses cgroups for resource limiting
- Assigns each app a unique UID / GID
- Updates access on /dev entries
- Programs netfilter firewall

Every process other than init is an app

```
{
  "SchemaVersion": 1,
  "Name" : "CalendarApp",
  "ComponentId" : "2e66f232-bc24-4c35-af3e-87517715ea05",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "Capabilities": {
    "AllowedConnections": [
      "login.microsoftonline.com",
      "graph.microsoft.com"
    ],
    "Gpio": [],
    "Uart": ["ISU0"],
    "WifiConfig": true
  }
}
```

User mode / App model

Attack surface is reduced by removing features

- This not only helps our RAM limits but means one less feature we have to reason about from a security perspective
- We have no shell or user account management
- No kernel module support
- We ship a greatly reduced library set (9 total .so files in /usr/lib)

Resource usage is limited

- cgroups enables predictable RAM exhaustion and resource contention behavior
- An app cannot access any peripherals or network traffic without opting into those features

Future Work

Upstream some of our work that is applicable outside our product

- Some memory improvements and configuration support are wins for any platform
- File system work maybe generally applicable to other classes of embedded devices

More namespace usage and further isolation of resources

- There's a lot of security goodness to be had here if we can make it fit

Split up some large capability sets like CAP_SYS_ADMIN

- The Linux capability model is great until you need CAP_SYS_ADMIN…
- Some "low level" hardware features are designed to require ADMIN when not strictly needed

Revisit features that were "too big" to fit by refactoring to enable usage with less resources

- For example, looking at other LSMs

Many, many, more ideas than I can possibly fit on this slide…

Takeaways

Security and resources are at odds at times

- Features are designed for strong security, space considerations are secondary (if at all)
- Security features are often all or nothing – no way to take some benefit without paying for all
- Many security features depend on other frameworks like sysfs

There's a lot that works well

- Most core Linux features just work even with limited resources
- Most of the changes we had to do were small modifications to existing code paths

Improvements for the desktop / server space often benefit embedded & IoT as well

- The problems in the IoT space are far from unique

Let’s secure the future.

SECURED FROM THE SILICON UP