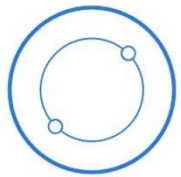


Azure Sphere: Fitting Linux Security in 4 MiB of RAM

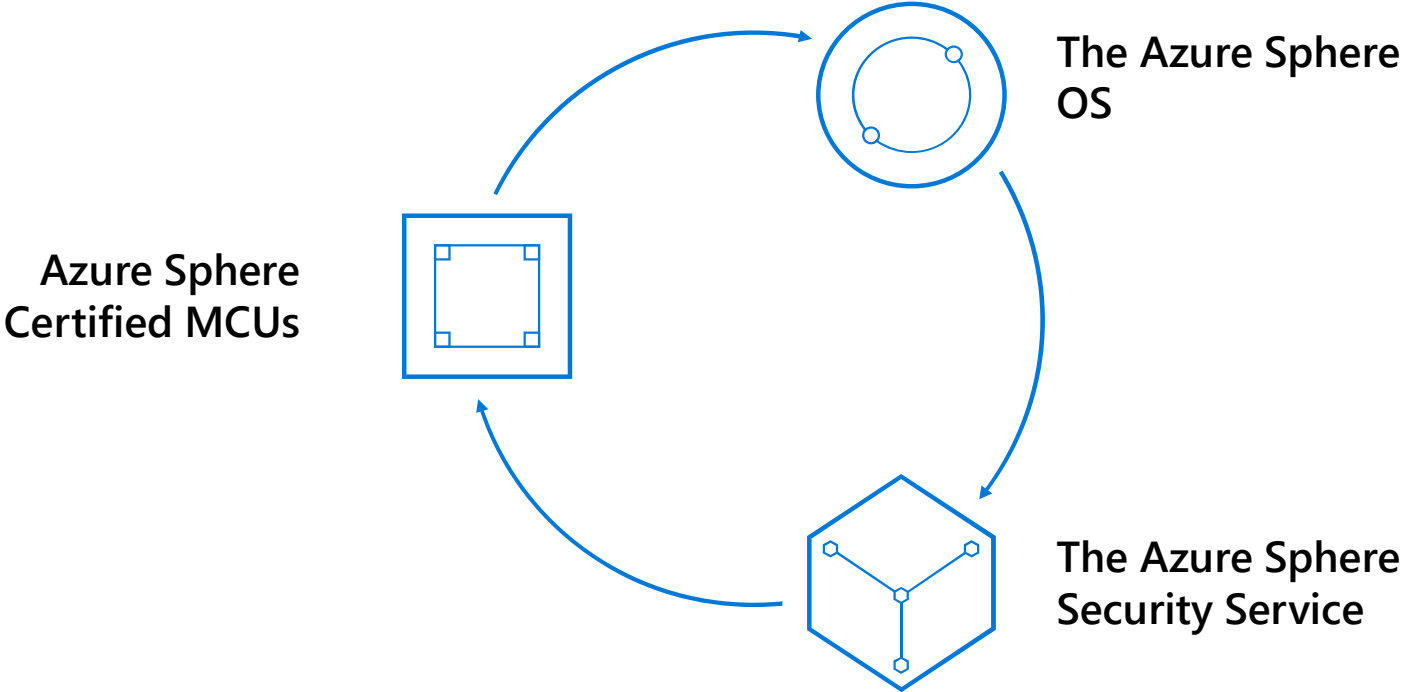
Ryan Fairfax
Principal Group Software Engineering Manager
Microsoft



Agenda

- **Intro to Azure Sphere**
- **Azure Sphere MCUs**
- **OS Architecture and components**
- **User mode services / App Model**
- **Future work / Takeaways**

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



The Azure Sphere OS

Secure Application Isolation

Compartmentalize code for agility, robustness & security

On-chip Cloud Services

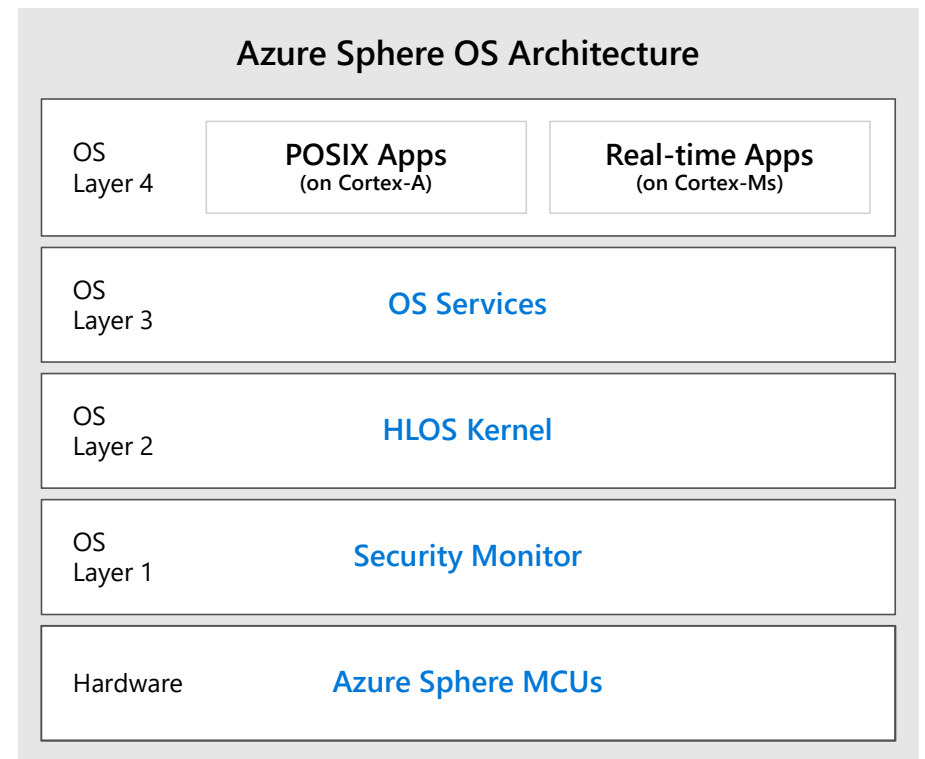
Provide update, authentication, and connectivity

Custom Linux kernel

Empowers agile silicon evolution and reuse of code

Security Monitor

Guards integrity and access to critical resources

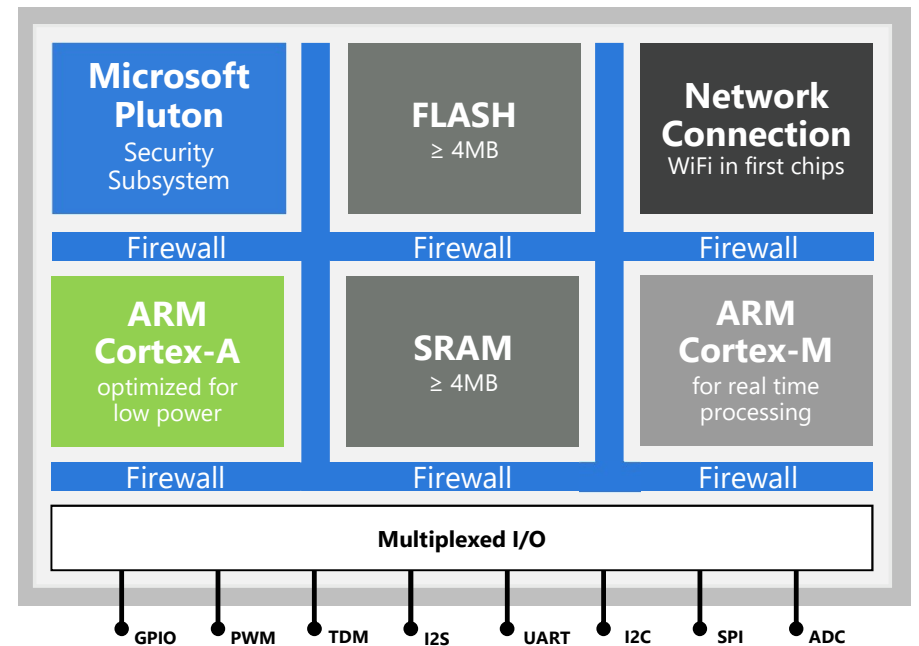


Azure Sphere certified MCUs

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



MediaTek MT3620

The first available Azure Sphere MCU is the MediaTek MT3620

- **2 dedicated CPU cores**
 - Cortex M4F – Pluton Security Subsystem
 - Andes N9 – 802.11b/g/n Wi-Fi
- **3 CPU cores for applications**
 - Cortex A7 @ ~500Mhz – Linux kernel, POSIX apps
 - 2x Cortex M4F @ ~200Mhz – RTOS or bare metal
- **Peripherals are available to every core**
 - UART, I2C, SPI, ADC, GPIO, PWM, I2S
 - System wide real time clock w/ alarm
- **4MB of SRAM, 16MB of NOR flash in package**

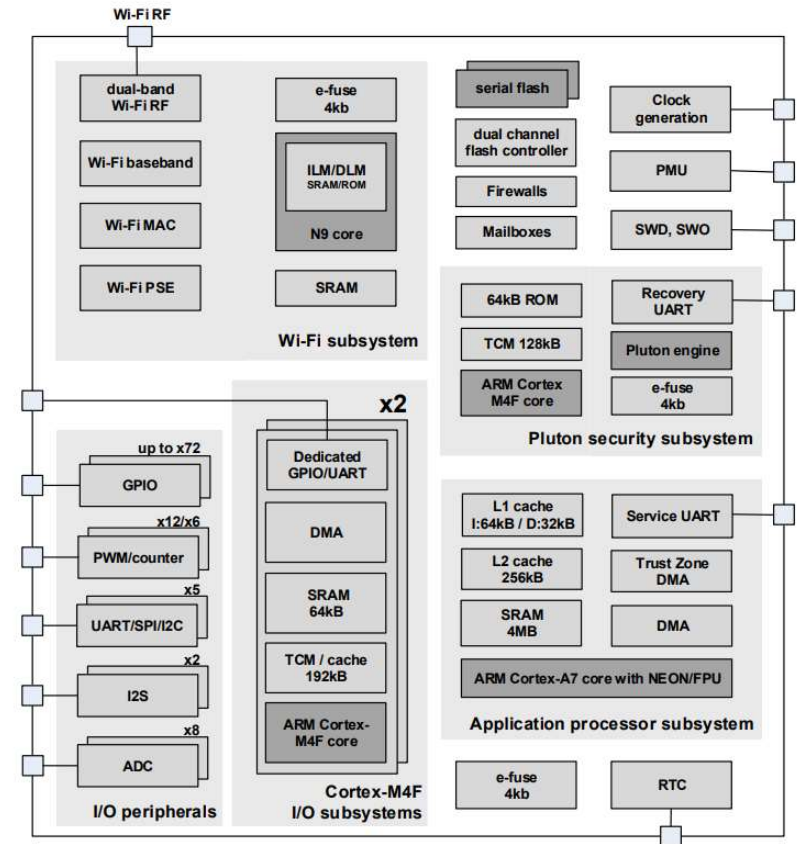
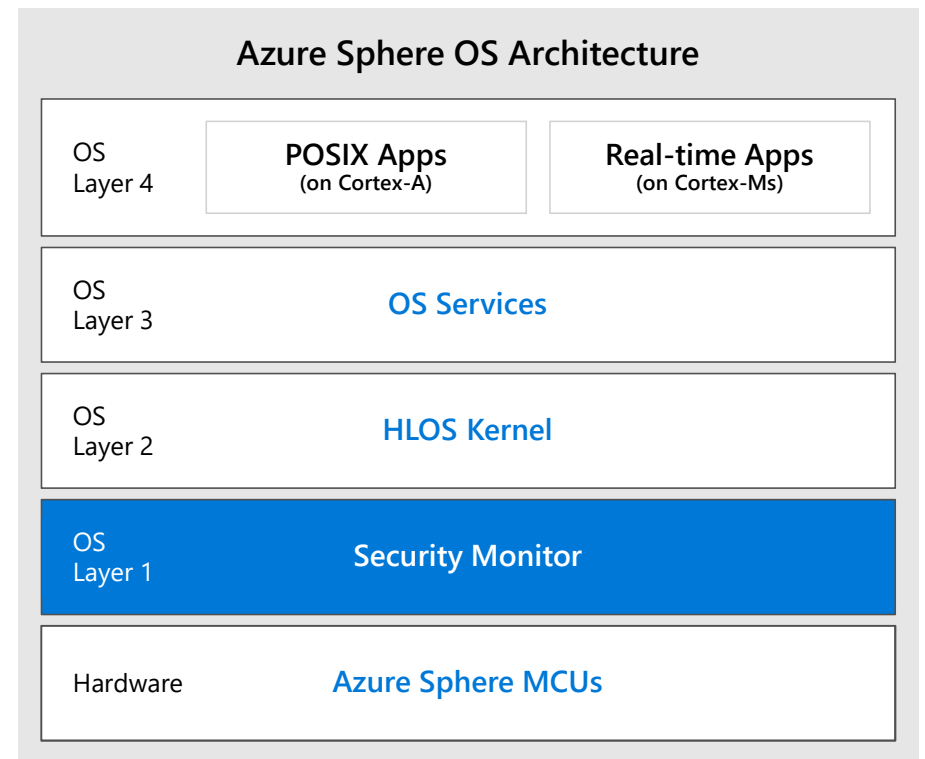


Figure 1. MT3620 Block Diagram

Security Monitor

The Security Monitor is the Trusted Computing Base of the OS

- Runs code critical to the security of the overall system
- Code is layered for defense in depth: some code on a dedicated core, some in TrustZone
- Kept as simple as possible to reduce attack surface



Pluton Security Subsystem

The Pluton Security Subsystem is the most trusted entity in the system

- Runs on a dedicated CPU core (Cortex-M4F on MT3620)
- Runs custom bare metal firmware (no threads, minimal interrupt usage, no apps)
- Pluton is the boot core in the system
 - Drives secure and measured boot
 - Hands off execution to Cortex-A core
- Responsible for highly security sensitive operations
 - Device authentication and attestation
 - Power management
 - Image verification
 - Trusted key management

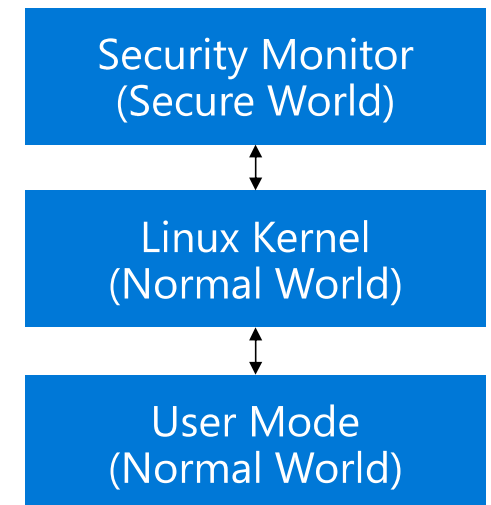
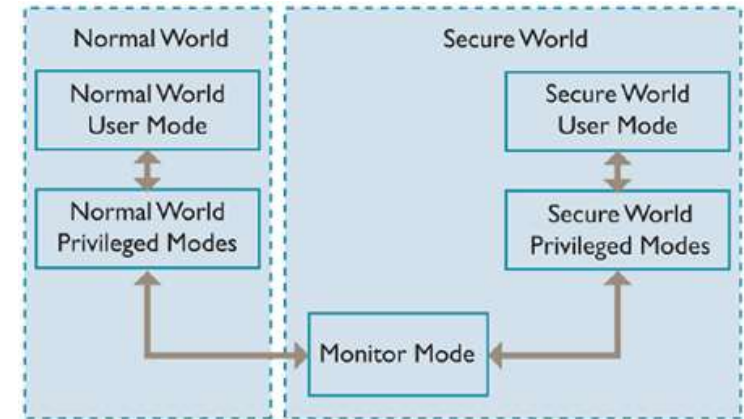
TrustZone Security Monitor

The other part of the TCB runs using ARM's TrustZone technology

- Shares CPU with Linux
- Context switch in and out of TrustZone based on interrupts and calls from Linux Kernel
- Only one mode is executing at any point in time

Protects critical resources

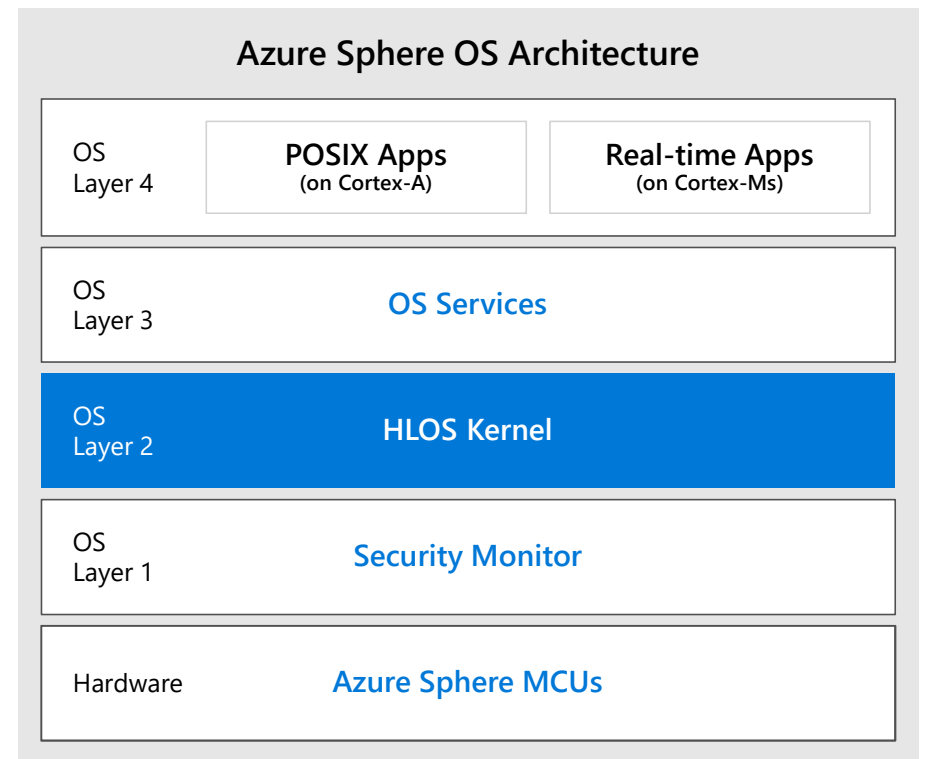
- Peripheral access control
- Update engine
- Writes to flash (but not reads)
- Divides up SRAM for use by various components



Linux Kernel Customizations

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

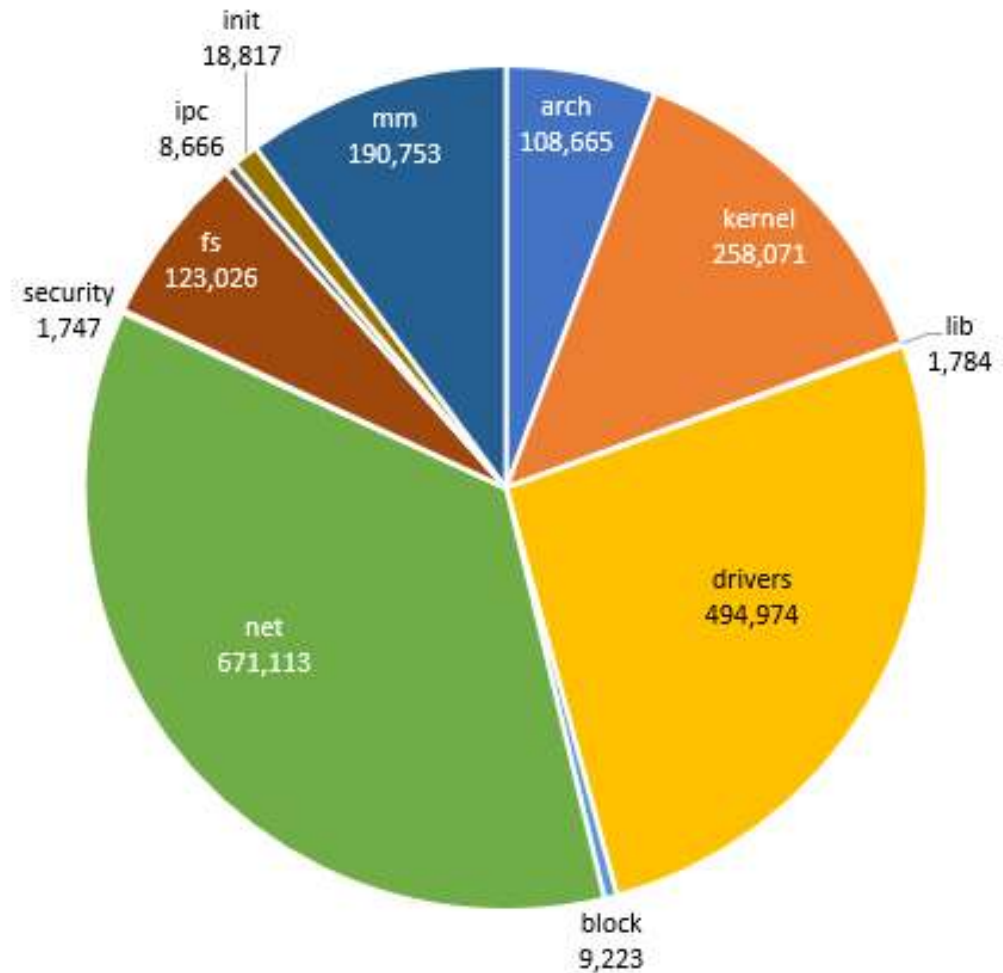
- Based on kernel.org sources
- Originally 4.1, now 4.9, with the goal to keep moving forward as LTS branches get declared
- Upstream releases are merged monthly
- We have ~350 commits in a branch on top of the upstream sources



Linux Kernel Customizations

The first challenge was making it fit

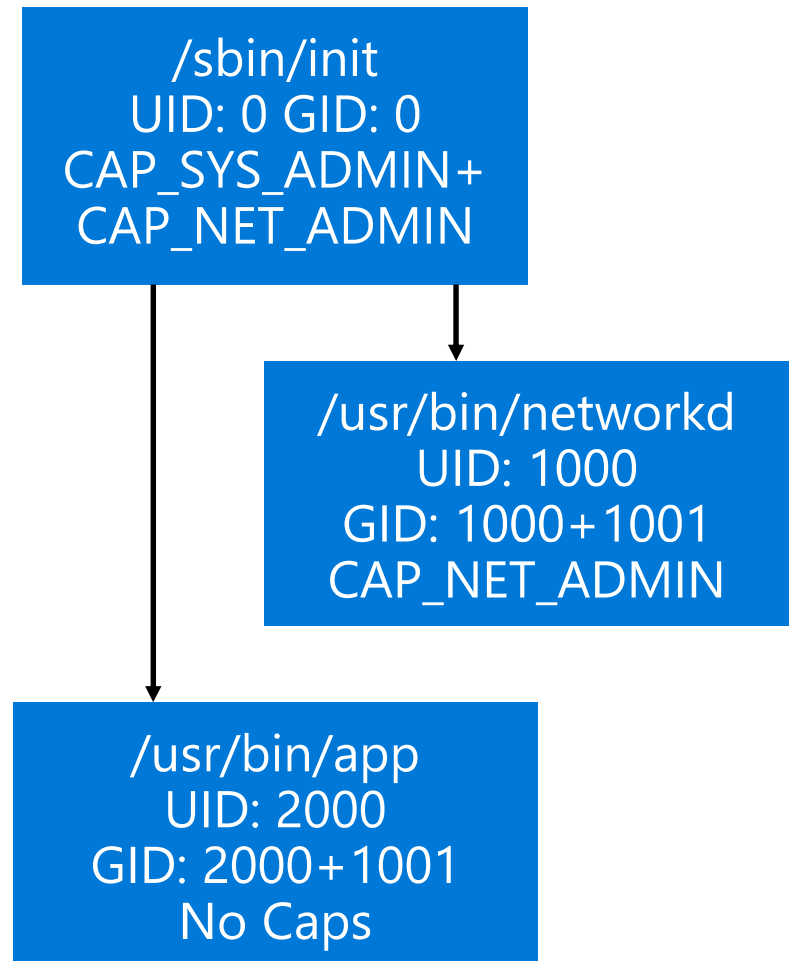
- The first build was a 4MB kernel, which took up all of the RAM
- We moved to Execute-In-Place (XIP) which helped a lot but we still used nearly 4MB of RAM to boot
- Lots of our early patches were making things more modular and tuning sizes of caches
- Some things were removed to reduce size (sysfs, 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

```
/ $ cat /proc/27/attr/current  
CID: 48A22E96-D078-4E34-9D7A-91B3404031DA  
TID: 9eca6399-06ff-4bec-aaa6-6107eae14d74  
CAPS: 00000043
```

- 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>

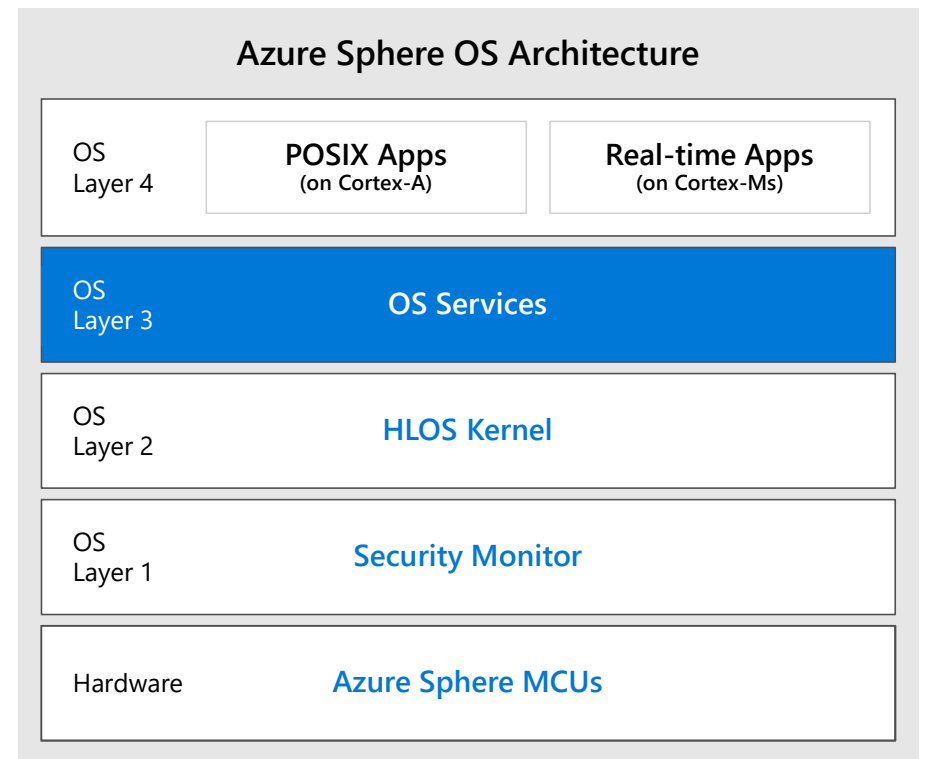
In some cases we added more access control to existing features

- 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

We provide several user mode services to support app development

- Includes libraries, daemons, development tools
- Leverages existing OSS solutions (musl, curl, etc)
- Customized where needed to reduce attack surface or satisfy size constraints



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

application-manager (init)

Network Services

Command and Control

Update Services

HW Crypto / RNG

GDBServer
(optional)

OEM apps

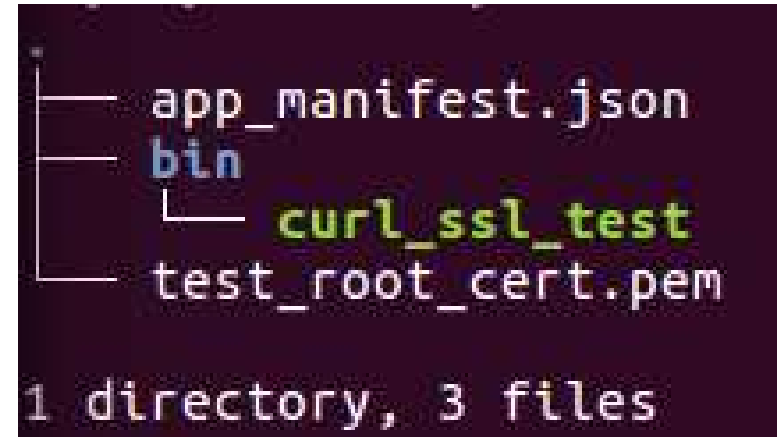
User mode / App model

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 must 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

Conclusion

Check out our documentation and samples:

- <https://www.microsoft.com/azure-sphere>
- <https://github.com/Azure/azure-sphere-samples>

Q & A