Poster: Design Considerations for Persistent Storage on Multi-Tenant Embedded Systems

Anthony Tarbinian
atarbini@ucsd.edu
UC San Diego
La Jolla, CA USA

Tyler Potyondy
tpotyondy@ucsd.edu
UC San Diego
La Jolla, CA USA

Pat Pannuto
ppannuto@ucsd.edu
UC San Diego
La Jolla, CA USA

Abstract

As microcontrollers become more capable, embedded systems are taking on increasingly complex responsibilities. As this trend continues, these embedded systems are shifting towards multi-tenancy. It's common to see these applications include multiple components from untrusted parties, running simultaneously on the same hardware. This poster discusses the challenges of designing an interface for persistent storage on a multi-tenant embedded system. In addition, it examines a minimal implementation and evaluates its successes and shortcomings.

CCS Concepts

• **Computer systems organization** → **Embedded software**; • **Security and privacy** → *Operating systems security*.

Keywords

Embedded OS, Security, IoT, Non-Volatile Storage

1 Introduction

Multi-tenant systems have unique resource sharing requirements: they must virtualize and fairly share often limited resources while also maintaining confidentiality between users. Historically, embedded systems have been single tenant. They are usually relatively simple and deployed for single-purpose applications. Recently, more embedded systems are becoming increasingly complex and see the need for multiprogramming. With this new reality, these systems are becoming multi-tenant as developers incorporate more third-party components to run on these devices.

This poster will focus on the ideal interface for persistent storage on multi-tenant embedded systems. It will put forward the questions facing designers and explore how existing persistent storage systems handle them. Then we will dive into how those questions are answered in the context of a minimal implementation for a secure embedded operating system: TockOS.

2 Background

When it comes to persistent storage, many embedded systems provide no isolation at all. This might be adequate for single-tenant systems, where the entire platform is known and trusted.

On multi-tenant systems it is desirable to ensure confidentiality of sensitive data across applications. In the context of persistent storage, this means that each app should not be able to read or modify another app's persistent storage. In this multi-tenant environment, each app is an untrusted entity. They could be written by third-parties and be potentially malicious or buggy. Say that there are multiple apps running on the same board. One app might save sensitive cryptographic keys so it can retrieve them across reboots. If all apps have unrestricted access to flash storage, another app could read that key, breaking confidentiality and opening the door for a whole class of malicious activity.

3 Design Considerations

When setting out to implement such a persistent storage system, a few questions must be answered. This section will explore these concerns and examine how many traditional desktop and mobile operating systems handle them.

3.1 App Interface

When designing a storage system, it is crucial to have an interface that is intuitive while still providing the necessary functionality. The designer must decide the best way to have apps interact with locations in storage. Single-tenant embedded systems might only use a barebones interface where apps can directly refer to physical storage addresses.

Many operating systems organize their physical storage into a hierarchical filesystem. From the app's perspective, they can open a file, seek to an offset, and start reading or writing. This abstraction also allows for apps to organize logically related information in separate files. All the while, don't have to worry about where their data actually gets physically stored.

3.2 Isolation Model

On multi-tenant systems, it's crucial to pick the right level of isolation to fit the system needs and threat model.

On UNIX-based operating systems, the isolation barrier is defined between users and groups instead of processes. Each file or directory has permissions that dictate which user owns it and how much permission various entities on the system have. This provides a great deal of flexibility for defining exactly what each entity can do with each file.

Some platforms take this a step further and provide a stronger isolation model using sandboxing. The iOS mobile platform has a filesystem, however it only allows apps to access files that are within that app's sandbox. [1]

3.3 Retention Model

Once a system has determined how to store and update its persistent storage, it must attempt to answer the question of when it's safe to delete data that isn't deemed necessary. On some systems with limited storage space, it might make sense to occasionally clean out storage space or move it somewhere else.

On desktop and server classes of computers, storage is usually plentiful and the system will not start clearing user files on its own.

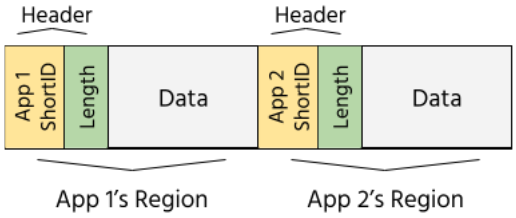

Figure 1: Layout of multiple app regions in persistent storage

Storage is only cleared with explicit user permission to delete the data. Mobile devices still mostly stick to this paradigm; however, their storage space can be much more limited. To combat this, they might occasionally offload on-device data to a form of cloud storage. One example of this is iOS's "Optimize iPhone Storage" which will upload larger, high resolution photos to iCloud while keeping small, low resolution thumbnails on device. [2]

4 Minimal Implementation

For TockOS, we've implemented a persistent storage interface which provides strong isolation between apps. As mentioned, Tock is a multi-tenant system where each "tenant" is not trusted. This section examines how this implementation addresses the previous questions and where its shortcomings lie.

Apps are permitted to perform read and write operations starting at a specified offset address and continuing for a given length. Apps use logical addresses, meaning their storage starts at address 0 regardless of where their data is physically located. This logical addressing reduces complexity for apps since they don't need to worry about where their data is relative to other apps'.

With this implementation, we've chosen to split up the physical storage space into fixed-size regions that exclusively belong to a single app. Unlike what most filesystems do, each app can only have ownership of a single region. The main limitation of this design is that apps cannot grow their storage size. They are given a single fixed-size region and have no ability to resize it. This decision was made to limit complexity of this initial implementation. Future implementations, should consider adding an interface for apps to request more space.

With that being said, this fixed size can be customized in the kernel at compile-time to allow for larger regions. Note that changing the size only is reflected for new regions that are created afterwards. Any existing region will keep its previous size. To accommodate potential discrepancies in region size, a header is written to the start of each region to describe its length as shown in Figure 1

When it comes to isolation, the Tock kernel enforces isolation between regions on every read and write. Apps are not trusted and therefore should not be able to access another app's persistent data. To enforce this, it uses each region's starting and ending addresses to perform the appropriate bounds checks.

One challenge in the implementation process was preserving ownership data across reboots. When the kernel boots up, it is unware to which apps had previously owned the storage regions. To aid the kernel, ownership data is stored in the header of each region as shown in Figure 1. Specifically, a 32-bit value called a "ShortID" is written to this header. A ShortID is assigned to each app by the Tock kernel and it carries the property that it is persistent across reboots. [3]

With regards to data retention, this implementation does not make an attempt to clear out existing data. While this means that no data will be lost or deleted, it does prove problematic as space fills up over time. If a user removes apps from their board and flash new apps, those old app regions will not get automatically removed and new app regions will be allocated to take up even more space. This problem is exasperated with the extremely limited storage capacity of embedded systems.

For embedded devices which are expected to run for long periods of time, it might make sense to rethink the data retention policy. While thinking about this problem, one idea that was brought forward was to present a programmatic interface for apps to give up their own storage space. While this would be a simple solution that could solve the issue of limited space, we thought that it wouldn't be a common enough scenario to be useful. Most apps using persistent storage will just save some data and let it sit there forever. There usually isn't a case where they will delete their own data and give it up to another app. Another approach could be to clear inactive app regions that haven't been accessed for some period of time. Once a region is cleared, it could be handed off to any other app that requires storage space. This requires apps to consent to their storage being cleared, which might not apply to all use cases.

5 Conclusion

In the case of persistent storage for TockOS, the "right" abstraction is still yet to be realized. What currently exists, is a simple implementation that guarantees isolation. However, it is limited by the fact that app regions cannot be dynamically resized and that there is no mechanism to free up existing regions. With future updates guided by discussion and community insight, we can iterate upon this minimal implementation to meet the needs of such a multi-tenant embedded system.

References

[1] Apple Inc. 2018. File System Basics. https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html. Accessed: 2024-09-20.
[2] Apple Inc. 2024. Manage your photo and video storage - Apple Support. https://support.apple.com/en-us/105061. Accessed: 2024-09-20.
[3] Philip Levis, Johnathan Van Why. 2021. TRD AppID - The Tock Book. https://book.tockos.org/trd/trd-appid. Accessed: 2024-09-20.