Project Overview

The goal of Project Ellesmere is to streamline identity verification in New York airports by installing systems that read/verify mDL data by installing physical systems equipped with the ability to read/verify mDL data. The data transfer is executed via an ISO 1803-5 compliant NFC/Wifi-Aware handoff sequence. The data is decrypted and verified by the verifySDK developed by the India team and then re-encrpted. The verification results are then sent to the CAT machine through a USB cable to be reaffirmed by a TSA official.

Driver Function

The function of the CAT-side usb driver that I've written is to enable effective USB serial simplex communication and data transfer between the dongle (which contains NFC/high bandwidth Wifi-Aware supporting chips) and the CAT machine (which is the host). I implemented the Android Accessory Protocol using the low-level API of usb4java, a java wrapped C library, for host-side operations and android.hardware.usb classes for device-side operations. Unfortunately, I was unable to successfully transfer data from the PC to the Hikey board as my testing was cut short by a hapless corruption of crucial Hikey system files which disabled adb over usb.

I have written a few drivers that differ in style and minor implementation details. A procedurally written driver is located under the serialization branch with filepath: host\procedural. There are two object-oriented drivers under the serialization branch with filepath: host\oop\async and host\ooop\sync, respectively. The former transfers data in an asynchronous manner while the latter transfers data in a synchronous manner and is the closest to completion.

Dongle-side App

The dongle-side application is located under the master branch under the device folder. It is triggered once the hikey is in accessory mode and uses a FileInputStream and FileOutputStream object to read/write data from/to the windows machine.

Data Serialization

In terms of data serialization, I read a fair bit about Google’s protocol buffers. Early on, I wrote a fairly naïve but functional program that transforms java objects directly into protobuf generated classes, saving the time of manually writing a .proto file. The program only works for simple cases but can handle nested objects(objects with objects (with objects… and so on) as data fields) complete with proper indentation. This program is located under the serialization branch at: serialization\protobuf\testing\objproto and is named ProtoConverter.java. The accompanying files are output, and templates.

A Maven build automating the compilation of .proto files is located under the serialization branch at: serialization\protobuf\testing\mvngooglex\autoprotoc. It uses an open-source plugin.