Simulate a smart card

I. Host card emulator basic

Android 4.4 and higher provide an additional method of card emulation that doesn't involve a secure element, called *host-based card emulation*. This allows any Android application to emulate a card and talk directly to the NFC reader.

Card emulation with a secure element

When NFC card emulation is provided using a secure element, the card to be emulated is provisioned into the secure element on the device through an Android application. Then, when the user holds the device over an NFC terminal, the NFC controller in the device routes all data from the reader directly to the secure element. Figure 1 illustrates this concept:

Figure 1. NFC card emulation with a secure element.

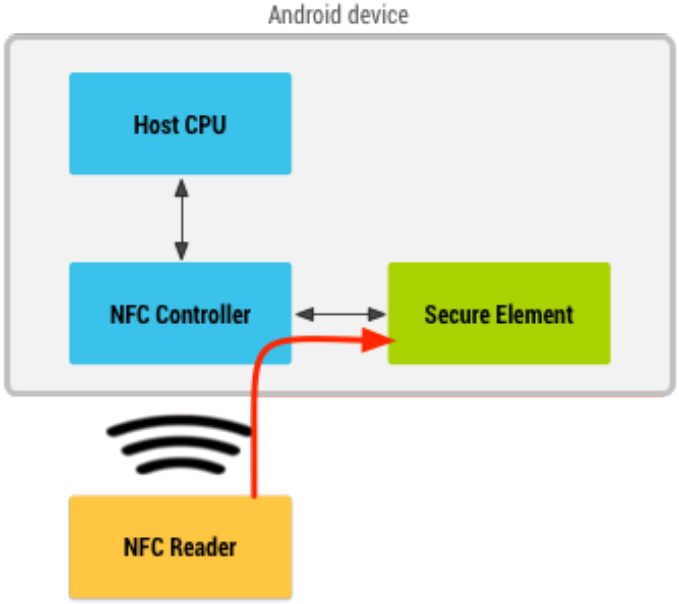


Figure 1. NFC card emulation with a secure element.

The secure element itself performs the communication with the NFC terminal, and no Android application is involved in the transaction. After the transaction is complete, an Android application can query the secure element directly for the transaction status and notify the user.

Host-based card emulation

When an NFC card is emulated using host-based card emulation, the data is routed directly to the host CPU instead of being routed to a secure element. Figure 2 illustrates how host-based card emulation works:

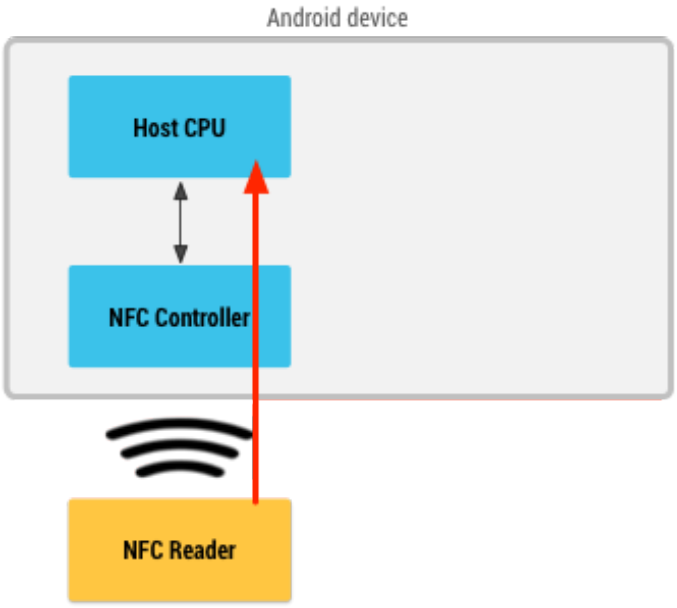


Figure 2. NFC card emulation without a secure element.

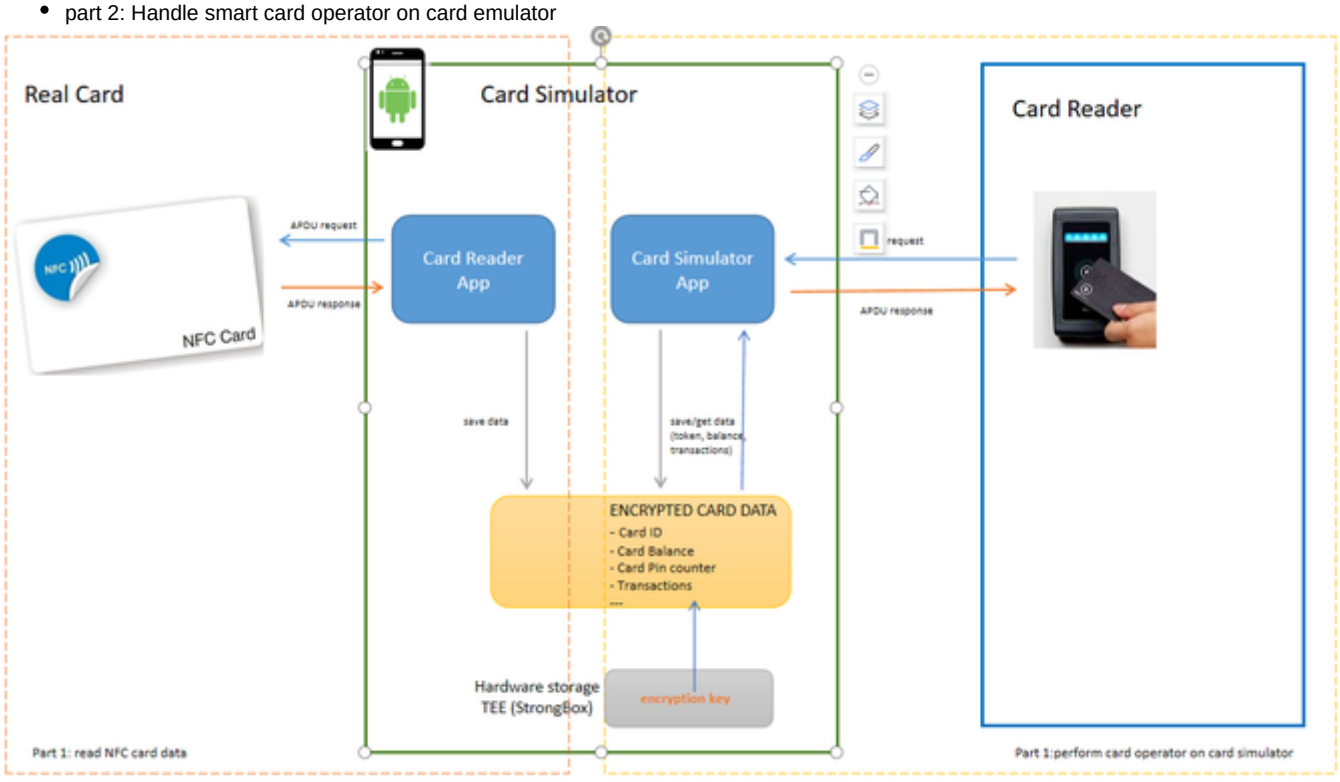
II. How to simulate a simple smart card?

Now we try to simulate a smart cart emulator, in this example is VISA DEBIT card.

1. Define what we need to be handled?

There a 2 main parts we need to be handled:

- part 1: Read the NFC card (real card) data: prepare data for HCE
- part 2: Handle smart card operator on card emulator



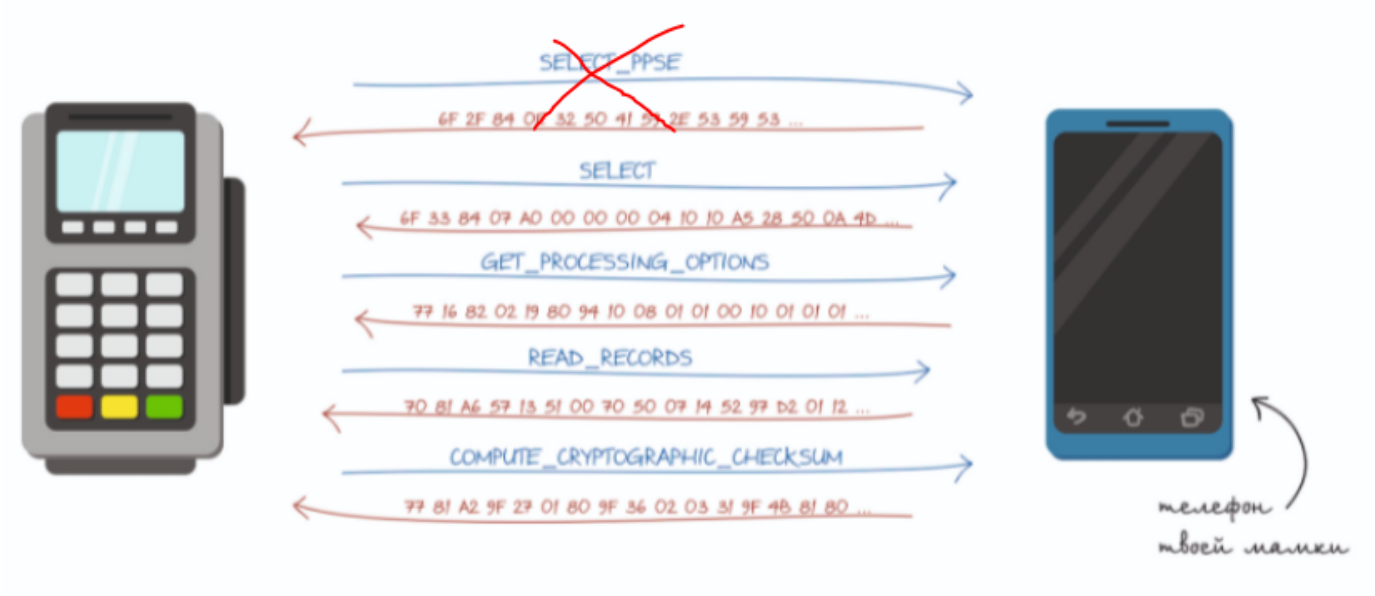
***a/* Read the NFC card data**

- Read real card data: for how to read the Visa Credit card, we can read https://smartdevllc.atlassian.net/wiki/spaces/VP/pages/2521399346/How+to+read+a+Credit+card+via+NFC#Read-the-Credit-card-via-NFC
- Storing card data: after reading the card, we can store the necessary information. So which information need to be stored? It depends on what we need to do with the NFC card. For example, in the case of a Visa Debit card, we need to store the SELECT_AID response, GET_PROCESSING_OPTIONS, READ_RECORD response, pin counter,... (or in case of VeryPay Offline, it may be card issue date, balance, transactions, ..)

The data should be store in a secure location, using TEE or StrongBox, and also being encrypted.

b/ Handle smart cart operator on card emulator device

From the *read a Credit card via NFC*, we know the basic communication between the NFC Reader and the NFC card.



the Smartphone now replaces the NFC card to communicate directory with the Card Reader, and because we already know exactly the AID needs to be selected, so the first step *SELECT_PPSE* could be removed

To simulate a smart card, we need:

- **HCE Service**: to make the android phone able to listen to the APDU command, we need to implement a service to handle this task. Android provides `HostApduService` that helps to do it.

```
class MyHostApduService : HostApduService() {

    override fun processCommandApdu(commandApdu: ByteArray, extras:
Bundle?): ByteArray {
      // this method is called whenever a NFC reader sends an APDU
command
        ...
    }

    override fun onDeactivated(reason: Int) {
        ...
    }
}
```

also declare this service in the *AnroidManifest* file

```
<service android:name=".service.HostCardEmulatorService"
    android:exported="true"
    android:permission="android.permission.BIND_NFC_SERVICE">
    <intent-filter>
        <action android:name="android.nfc.cardemulation.action.
HOST_APDU_SERVICE"/>
    </intent-filter>
    <meta-data
        android:name="android.nfc.cardemulation.host_apdu_service"
        android:resource="@xml/apduservice"/>
</service>
```

- **Define AID**: We know each NFC card type has its unique AID. This id is used when the Card Reader executes the SELECT AID command to get the basic card data, it gets the necessary card information to perform another action.

For example, with a VISA Debit card, the aid is `A0000000031010`

Note: If we implement our own NFC card, change this aid to our aid

After we have the AID, add this into the `aid-filter`. If the Smart card Simulator app has registered with the `aid-filter` so that if the Card Reader sends an APDU command to SELECT AID, the Android OS will point to the registered simulator app.

```
<host-apdu-service xmlns:android="http://schemas.android.com/apk/res
/android"
          android:description="@string/servicedesc"
          android:requireDeviceUnlock="false">
    <aid-group android:description="@string/aiddescription"
              android:category="other">
        <aid-filter android:name="A0000000031010 "/>
        <aid-filter android:name="F0394148148100"/>
    </aid-group>
</host-apdu-service>
```

- **Handle APDU command**:
  - Define the ADPU command list that needs to be handled
  - Define the action with each APDU command
- **Storing the Card data**:
  - get the card data to verify the APDU command
  - store/update the card data (balance, transaction, ..)

2. Implement a simple smart card (VISA card)

Let start to implement a very basic app that simulates a VISA Debit card. This app has 2 main modules:

a/ Card Reader:

a1/ read a real NFC card.

the data displayed on the screen after reading a card is just the original response from the card.

- add NFC permission to the manifest

```
<uses-permission android:name="android.permission.NFC"/>
```

- implement the `NfcAdapter.ReaderCallback` interface and override the `onTagDiscovered` function, it's called each time an NFC card connects to the Card Reader. This is a place where we perform the read card functions.
- connect to the `NfcAdapter` and manager to `enableReaderMode` nd `disableReaderMode`
- define APDU commands: This module just performs some basic commands. Because this app only targets one type of smart card, so we can define hardcode the APDU commands:

```
val apduCommandMap: Map<String, String> = mapOf(
    "SELECT_AID" to "00A4040007A000000003101000",
    "GET_PROCESSING_OPTIONAL" to
"80A8000023832127800000000000000001000000000000079200000000000940211005007AD0011C00",
    "GET_PIN_TRY_COUNTER" to "80CA9F1700",
    "GET_TRANSACTION_COUNTER" to "80CA9F3600"
)
```

- perform read card data and store it into the device storage (TEE).

```
val preferenceProvider = PreferenceProvider(applicationContext)
for ((step, apduCommand) in APDUUtil.apduCommandMap) {
    val response = APDUUtil.executeApduCommand(apduCommand, isoDep)
    preferenceProvider.putString(step, TypeConvertor.toHex(response))
    result += "\n${step}: " + TypeConvertor.toHex(response)
}
```

a2/ read a “card simulator” to get the card data.

No need to do more extra work because the things we did above are enough.

b/ Card Simulator:

simulate a VISA Debit card, works as an NFC card.

b1/ App and service config

- register the *hce* feature:

```
<uses-feature android:name="android.hardware.nfc.hce"
    android:required="true"/>
```

- register a Host card emulator service to listen to the APDU command sent by the Card Reader

```
<service android:name=".service.HostCardEmulatorService"
    android:exported="true"
    android:permission="android.permission.BIND_NFC_SERVICE">
    <intent-filter>
        <action android:name="android.nfc.cardemulation.action.
HOST_APDU_SERVICE"/>
    </intent-filter>
    <meta-data
        android:name="android.nfc.cardemulation.host_apdu_service"
        android:resource="@xml/apduservice"/>
</service>
```

- defines the AID. If the Card Reader sends a SELECT AID command and the AID in this command does not find it in the *aid-filter* so the service will not handle this command.

Each NFC card type has its own AID (for example the AID of a Visa Debit card is `A0000000031010`) .

```
<host-apdu-service xmlns:android="http://schemas.android.com/apk/res
/android"
    android:description="@string/servicedesc"
    android:requireDeviceUnlock="false">
    <aid-group android:description="@string/aiddescription"
        android:category="other">
        <aid-filter android:name="A0000000031010"/>
    </aid-group>
</host-apdu-service>
```

b2/ Implement the Host Card Emulator service

- implement the `HostCardEmulatorService` extends the `HostApduService`

`class HostCardEmulatorService: HostApduService()`

- process the APDU command sent by the Card Reader: determine the APDU command type then return the equivalation response. get the card data from the TEE storage which is saved by the “card reader” above.

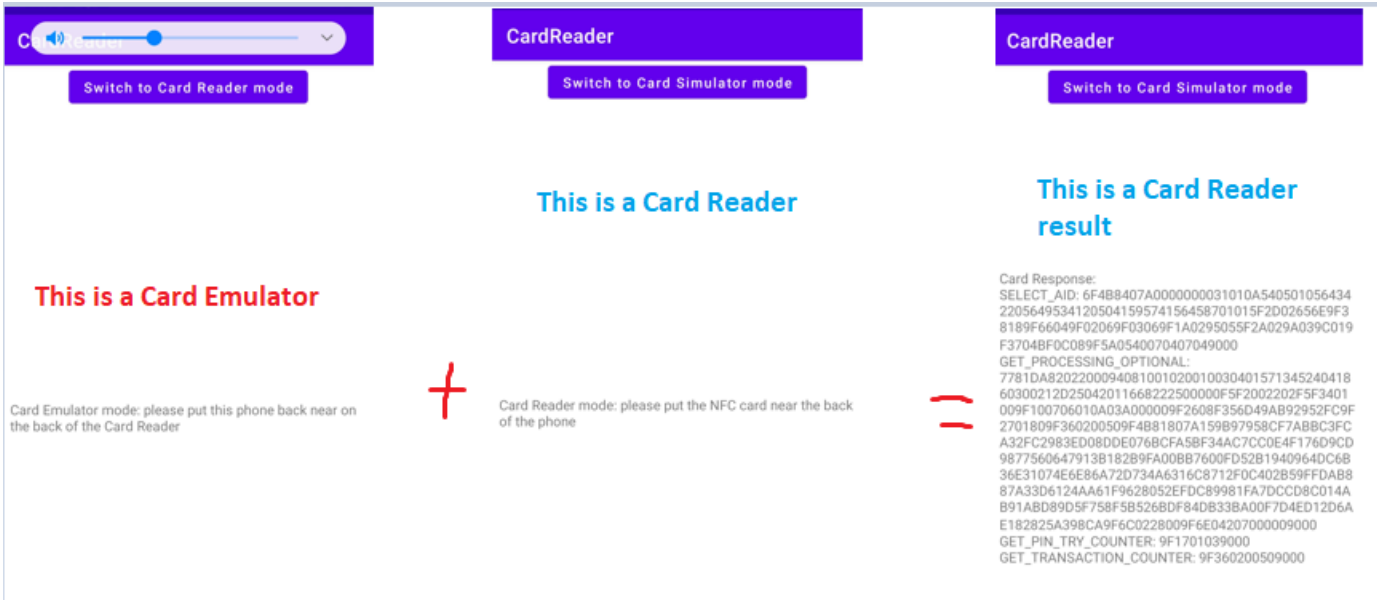
```
override fun processCommandApdu(commandApdu: ByteArray?, p1: Bundle?):
ByteArray {
    if (commandApdu == null) {
        return TypeConvertor.hexStringToByteArray(STATUS_FAILED)
    }

    val commandApduStr = TypeConvertor.toHex(commandApdu)
    val apduKey = APDUUtil.findAPDUKey(commandApduStr)
    apduKey?.let { key ->
        val apduResponseStr = mPreferenceProvider.getString(key, "")
        return TypeConvertor.hexStringToByteArray(apduResponseStr)
    }

    return TypeConvertor.hexStringToByteArray(STATUS_FAILED)
}
```

- also can update the card data if need (for example update the pin try counter, the balance, …)

Here is the result (it works the same as when we used the real NFC card



c/ Source code sample

ap sample