23/10 Install Java Card JDK and its dependencies, read specification.

24/10 Install and get accustomed to Eclipse, set up project, configure it with Java Card JDK. Created a new GitHub repository and initialised it with documents, JDK specifications, and code.

27/10 Install OpenCard framework and read documentation

28/10 Started working on skeleton code for host-side app using code samples and OpenCard api docs.

29/10 Wrote most of both host-side and card-side app, haven’t yet worked out how to install onto card because I don’t have my card reader.

31/10 Reader arrived. Read more of the documentation, built a project. Installed gpshell, installed reader driver, successfully loaded gpshell sample applet onto card, listed contents, and removed it.

2/11 Attempted to load project onto card, discovered that my java cards are not JCDK 3.0.5 compatible. Choice: Get more up-to-date java cards, or use the older build process with the older JCDK? Reverting to older process means no handy Eclipse plugin. Documentation is also much better in newer version. Decided to buy new cards.

4/11 Attempted to run sample project by running OCF host app and connecting to card. OCF wouldn’t recognise the card. I think I’ve identified that the reader needs an OCF driver, but it only has a PC/SC driver. Sources are few and far between. May have to redo application in C++ to use PCSC API. Could also be incompatibility with Windows 10, perhaps try in Ubuntu first.

Data sheet for reader only specifies PCSC whereas for some other readers their data sheets also mention OCF.

From OCF programmer’s guide: “The OpenCard Reference implementation comes with a lockable CardTerminal implementation for PCSC card readers”. Downloaded reference implementation, there’s source code for PCSC-related stuff. Pcsc-wrapper-src.jar contains class files. Should look into usage. Very little information online about them. In particular, contains Pcsc10CardTerminal which emulates OCF stuff on PCSC.

New cards arrived. Tried installing apps, got Unknown ISO7816 error: 0x6438 for apps compiled via eclipse. Possibly because card is jcdk 3.0.4 compatible, compiled using jcdk 3.0.5. Appears to be card-defined execution error.

5/11 Decided to give the Python library pyscard a try instead of OCF. Spent the day running into various problems to do with dependencies it couldn’t find. Turns out plugin only for python 2, but I was trying python3. Working now, but unable to establish context. Not sure what the problem was, but it appears to be that particular version (1.7.0), because it worked when I tried a different version. With it, I was able to successfully test a small program that sends a SELECT APDU to the ISD and prints the sw1-sw2 output.

The pyscard Python extension is likely to be my tool of choice going forward. It’s compatible with both Microsoft PC/SC and Linux PC/SC lite, so the code will be portable and relatively easy to write.

7/11 Followed through applet compiling process for JCDK 2.2.2. Problem running first demo with jcwde, gives message “card was unexpected at this time”. No idea what it means. TODO: Try another demo. Not a serious problem though, it’s the simulation test suite so it’s optional.

Compiled sample code to class file, attempted to convert to CAP file. Converter has message “card was unexpected at this time”. Wtf?

Identified. It doesn’t like spaces in the JC\_HOME environment variable. Had to change directory structure.

New problem: Script also can’t deal with space in path of JAVA\_HOME. Scripts involve something like %JAVA\_HOME%\bin\java -classpath %\_CLASSES% com.sun.javacard.converter.Converter %\*, where they should have quotes around the environment variable. Error on distributor’s part. Have to reinstall jdk into a different directory tomorrow.

08/11 Was able to solve the problems and compile and convert an applet, and store/remove it from the card. Used JCDK HelloWorld sample applet source file.

09/11 Looked into different asymmetric cryptography protocols.

11/11 Configured Atom with Python and Java IDEs so I could develop both ends side-by-side. Adapted the build process to work with my project structure so I could upload my own apps, not just sample apps. Was able to successfully select my applet on the card and send a message, the applet checked its CLA and INS values, and returned them. Now have a better idea of how applet selection works.

Wrote a small script to automate the process of compiling source code, converting to a .cap file, uploading it onto the card, and running it with a test host application.

12/11 Wrote a test application that takes a byte string via an apdu reading “Hello World”, storing it, and returning the string upon a later apdu request.

14/11 Read some standards on protocols, decided to implement OPACITY protocol. Researched various components e.g. EC cryptography, Diffie-Hellman etc. and read parts of NIST 800-56A which was referenced by OPACITY protocol.

15/11 More reading of standards. Started writing host-side skeleton code to plan the structure of the app, researching Python crypto plugins (using ecdsa plugin). Researched Basic Encoding Rules defined in ASN.1, rules for encoding data types for serialisation, necessary because opacity CVC values sent between card and host are in this format.

18/11 Implemented various helper functions for host app, including cv\_extract, hash, truncate8, verify\_mac, extract\_fields. Researched Basic Encoding Rules (BER) to understand the structure of CVC values send in the protocol. Learned more about Elliptic Curve Diffie-Hellman, experimented with python ‘cryptography’ library, much more extensive than ecdsa and other crypto libraries.

22/11 Implemented various auxiliary functions for card applet including hash, truncate, get\_secret, kdf, and wrote mostly-complete code up to the ‘Zeroize Z’ stage of the card protocol.

23/11 Discovered no CMAC function exists on Java Card 2.2.2 platform. Only implemented on Java Card 3.x. Had to write my own implementation of CMAC in order to stick to the protocol. Wrote about 100 lines to this end, AESCMAC128.java.

25/11 Finished writing CMAC function. Got a little confused on a couple of points so referenced another implementation <https://pastebin.com/JQ9xQ5vK>. Used this to write the rest of the code in the authenticate function on the card, covering all points in the protocol. Still things to be done, including how to initialise with a CVC, properly parsing the format of EC keys.

26/11 Made modifications to CMAC class. Fixed some errors in main opacity implementation, and cleaned up so it uses fewer temporary buffer arrays. Wrote basic host-side code for key issuing system. Decided the card will generate its keypair and send the public key to the issuing system, which formats CVC and sends it to the card in a separate APDU. Created new compile/install scripts.

Attempted to compile. Various errors relating to uses of, and indirect casts to, int instead of short. Fixed those. Successfully compiles and converts. GPShell install command returns 6A80 (wrong data / incorrect values in command data) Will fix tomorrow.

30/11 Fixed. Problem was likely due to install method not calling register. Kp.genKeyPair() not working, throws (6F, 00). Turns out it was because KeyPair constructor can’t take arbitrary key lengths, only the constants in KeyBuilder. In JC 2.2.2 they only go to 192b, but protocol requires 256b. Instead, have to separately initialise public and private keys using NIST EC parameters and use the other KeyPair constructor. After adding it in, found it didn’t accept the compressed G, so had to enter the full uncompressed version. Fixed this, now generated keypair without error. Had trouble with getW() command to obtain public key. After a while figured out 64B buffer not big enough. Need 65B to accommodate extra 0x04 at the beginning.

Am now reasonably confident key issuing process works. Will move on to debugging authentication process.

3/12 Fixed usage of hash function and other errors on the authentication code. Removed unnecessary array conversion functions and cleaned up code. Implemented various uncompleted functions on host side including ec\_dh, kdf.

Realised that assumption behind signature was wrong. Should be calculated by the card and sent to host. Adapted issuing code to account for this.

5/12 Implemented digital signature on card

6/12 Had curious bug with digital signature code on card. Throws some unknown exception. I’d assumed the signature would be 20B (implied by documentation), probably 10B r followed by 10B s. Instead, signature was 0x47B (71B). Actually, it just generates 20B SHA-1 digest, not an option for the protocol, have to develop my own ECDSA implementation with SHA-256. Did much of that today but need to multiply byte arrays. Found online JavaCard library JCMathLib that supports this as well as other things. Open-source, free to use as long as its license is distributed with the code. Familiarised myself with this library.

7/12 Implemented (untested) signing part of ECDSA algorithm on card. Could also implement verify for completeness but unlikely to be needed by my code.

9/12 Attempting to upload code including new Signature implementation onto card. Not working for some reason (load() returns 6A80: Wrong data / Incorrect values in command data.). Narrowed down to use of ECConfig in sign() method. It seems that, when loading onto the card, the card wasn’t able to locate the library. I assumed linking the exp file was enough but it appears the exp file only outlines the api of a module. Module provided didn’t come with exp file for the cap file so I initially made my own. Now I have to gen cap file along with exp file and upload the cap file. I did this with the AID of the original cap file.

Trying to compile without specifying applet, get error about ill-formed conversion. Delete the Applet classes, get error about not allowing static array initialization. Try compiling with applet, resulting export file doesn’t contain public classes… I’m pretty sure I need to convert as a library i.e. without an applet but the way the code is written doesn’t seem to work with that. I wonder if the author of this “library” even intended it to be used as a library at all, or just a tech demo…

Removed unneeded classes that initialize static arrays leaving one more, which initialises {0x01} and {0x02} as constants. Moved initialisations to existing initialize() method. Converted successfully. Resulting exp file had all needed data. Error uploading to card (6F00 unknown error). Ree.

My only remaining option may be to move the library code files into my own package but that may exceed the 64KB package limit.

12/12 Couldn’t compile the library. Have instead copied necessary source files into main package, which was successfully compiled.

Wouldn’t properly upload, returning 6A80 on load() (wrong data). Narrowed down to use of TLV things – Error triggered by “new PrimitiveBERTag()”. The likely reason is that the TLV package is a javacardx package which may or may not be implemented. The manufacturer likely didn’t implement it. Instead just returning 32B each of r and s, concatenated. Can implement BER later.

Problem with library function mod\_inv. Calls set\_size on helper Bignat object in ResourceManager helper\_BN\_F, which should have max size 129 bit has 67 instead. Need to find out if it changes at some point and when

15/12 Implemented modular inverse function myself. Untested.

17/12 Integrated modular inverse into signature generation ecdsa algorithm. Appears to work – no errors and returns the right amount of data. Correctness of output not yet tested. Signature only successfully generated the first time the function is called, which shouldn’t be the case. Fixed – problem was that ECConfig can only be initialised once.

Then had weird error that was thrown due to array allocation, only thrown the second time around. Deduced it was due to memory being used up. Stuck a requestObjectDeletion() at beginning of process(), which solved it.

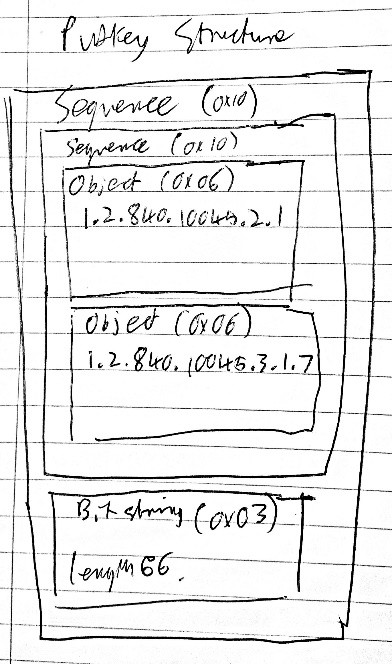
Then had library error code 0x7007, meaning lock object not found, thrown by mult function call within mod\_mult() Bignat function. Only thrown the second time. It appears the library maintains a list of lockable objects, one of which the library locks and unlocks on. The second time round, it can no longer find the same object to lock on. It seems the library has a bug to do with locking. Got around this issue by disabling object locking in jcmathlib. Now card appears to successfully generate public key and signature.

I appear to have broken my first card 😊 I guess they don’t like being reprogrammed hundreds of times. Good job I bought 4.

Error in set\_cvc function. Gets Lc and casts from short to byte, but when casting, the more significant bits are padded with 1s rather than 0s in Javacard for some reason. Fixed using Util.makeShort(). Now set\_cvc works.

18/12 With key generation now working, started working on authentication code. Python authentication code full of bugs and uncompleted bits because I didn’t know enough at the time to do it. Fixed most obvious bugs e.g. poorly formed APDU commands. Public key data generated in DER format, 91B long, rather than the 77B expected by the applet (and the standard).

After looking at the structure, it’s subtly different. Rather than a single object identifier, it contains a sequence of them, with two entries. The public key section is also different – instead of the format 0x04||X||Y, it uses 0x00||0x04||X||Y. Unsure of the reason why. To conform to the standard, I’ll need to extract the bit string field containing the key, and remove the first byte, before sending.

 (Bit string is in aforementioned format)

19/12 Modified card and host code to transfer 65B format public key, and began testing auth code. Appears to be a problem with generateSecret() KeyAgreement method. I was only giving it the x coordinate but it wants both in format specified in ANSI X9.62 section 6.4. As it turns out, this is the format initially given by the Python library.

Tried this, still failed. Threw ILLEGAL\_VALUE Cryptoexception. Means either “publicData format is incorrect or is inconsistent with PrivateKey specified during initialization”. Not sure why, could be because one of the specified object identifiers is wrong. I’ll try using only the second one and checking the point representation in 4.3.6 of X9.62. The first byte of this appears to be 0x04 rather than 0x00 so I’ll still have to truncate the first byte.

20/12 Still getting ILLEGAL\_VALUE error. Not sure why. Decided to implement it myself since it’s not very complicated.

While doing this, I noticed that I’d forgotten to uncomment some code in issuing process. I tried it and it failed. I’ll need to debug this. It appears that, again, it fails because too many arrays are allocated. I need to reduce memory usage somehow. Done this, now it seems to work again.

“Unknown error” in authentication process after having replaced ECDH. Seems to be issue with hash function in KDF. The byte array used to store the keying material isn’t big enough to store the last iteration. It did 3 iterations generating 32B each time, 5\*16B was needed, so the last hash only needed to produce 16B so the 32B didn’t fit in the array. Good now.

Problem in cmac function. Error is in the subkey generation, in the array left-shift.

23/12 There were also structural bugs in CMAC in both the update and sign functions. Fixed I think. Now works without throwing an error, authentication returns data. Total time taken is consistently around 2.9s (Needs improvement)

24/12 Now on host side it has issues deserializing a DER key in the Issuer formatted by the Issuer. The library function for deserialising keys expects a DER structure defined by RFC5280, which is quite complicated and defines a full certificate, which is not relevant. I’ll write my own implementation.

25/12 Implemented ecdh, fixed bugs in python code overall. Now runs to completion to the point of verification, but the mac verification fails. Problem needs debugging.

27/12 Walked through to establish where it went wrong. First checked card ID, which was different on card to what was calculated on host. This was due to the CVC hash output being different. When hashes returned were compared to hashes generated directly from the previously sent CVC, they matched, so it was an issue with how the host handles the CVC. Seems to work now.

Both ends know each other’s pubkeys but shared secret isn’t the same. For some reason, card’s privkey is 32B but host privkey is 109B.

28/12 Found out that reason was that private key octet string in PKCS8 is itself a DER structure. Decoded to get 32B private key.

I tried getting key values for card and host for one iteration and running both ECDH instances on the host to see if it works. It didn’t.

For some reason I was unable to even verify that for a keypair, Q = dG.

29/12 Emailed Markus for help since I couldn’t figure it out. In the meantime, I assumed both ends successfully calculate z and tested kdf.

Realised I forgot to initialise info input. Done.

While testing, I realised that the card returns Z as a 64B value (I had previously assumed it was 32B). This may be related to my problem. In this case, the KDF needs to generate more bytes to cover all of NextZ.

Tested KDF, seems to work.

Tested CMAC on both ends with same input, got different results. Used 3rd party library for host implementation so that’s probably correct, meaning there’s an issue with my implementation.

I should run the standard tests for cmac that came with the other implementation I found.

30/12 Discovered CMAC doesn’t work with input message of length 0. Fixed so it doesn’t fail, but still outputs wrong value. Problem with subkey generation, in the left-shift part. I didn’t realise that c << 7 pads with 1s. Another bug with it, I was initialising L as the key K rather than a block of 0s. Works now.

Various other issues requiring a restructure of CMAC. Now successfully does mac of empty message.

Doesn’t work with 16B message, output incorrect. Due to minor branch error. Now works for single 16B block.

Tested with other length messages. Now confident that CMAC works. Now only EC\_DH doesn’t work properly.

2/01/2018 Realised the reason EC\_DH wasn’t working is because I was using the traditional mathematical operators + and \* for EC point operations, not the special EC operators.

Libraries tend to use GMP (GNU Multiple Precision Arithmetic Library) in EC point operations. A couple of implementations I found that don’t were far too slow. I eventually found a decent free-to-use library. ECDH works on host now.

Now I know more about EC, I retried using the JC library for Diffie Hellman, but the output didn’t match the secret computed by the host, and wasn’t even the right number of bytes. No idea why, can’t find any information about it online. May have to implement it myself.

Attempted to use the JCMathLib library to do it, but there’s a problem with implementation of point multiplication. It attempts to the Java Card library KeyAgreement in a way incompatible with the JC2.2.2 I’m running (Algorithm number = 3). I’ll have to implement it myself (including the point doubling).

Appears to have an implementation of point add though, but mod\_inv Bignat function is proving to be a problem again – it’s called indirectly by the EC point addition and throwing the error I was getting on 15/12. Perhaps I’ll implement mult myself.

3/01 Started work towards implementing multiplication by implementing point add.

4/01 Big fixes in point add. Modified point add to also work as a point double function. (similar operations). Made it more memory efficient by adding intermediate Bignats that are reassigned instead of constantly allocating more BIgnats.

5/01 Implemented point double and then point multiplication, largely untested but outputs of add and double don’t appear to match the python library output for those functions.

6/01 Appears to be a problem with the modular inverse function I wrote previously. Decided to restart from scratch, more closely following more reliable sources. In implementation of egcd it was failing at a multiplication during the 13th pass for some reason. Throws some CardRuntimeException.

Frustratingly, all my cards have bricked. It seems that if a card fails too many times it is permanently deactivated. I’ve ordered more and am looking at the java card simulated runtime environment.

7/01 Setting up Java Card WDE and configuration file jcwde.app. Launch runtime environment with “jcwde simulation.jcwde.app”, listens to APDUs in T=1 on port 9025 (contact) and 9026 (contactless) by default.

14/01 Took a break for a week to finish off other work and move back into college. Error in JCMathLib implementation of Integer modulo I think – just takes modulo of magnitude. Overall too complicated. I wrote my own more basic implementation of big integer type to reduce possible points of failure. Still fails on the 13th pass, but at the second multiplication rather than the first. Suggests there may be some kind of memory leakage. Tried calling garbage collection but that didn’t work, possibly because the assigned memory is never not referenced. It seems it takes 2 consecutive failures to brick a card.

It seems I am at an impasse. I can’t use the KeyAgreement function because it doesn’t seem able to handle 32B EC keys. I can’t write my own implementation because of this curious bug that may be caused by the 3rd party library I am using. If I tried to write my own implementations of the library functions it would take too long, stray too far from the purpose of the project, and wouldn’t be anywhere near as efficient because the library uses hardware tricks I don’t know about. This single function has also killed 6 cards so far at a cost of around £25. I can’t continue like this.

I tried switching to the newer cards I have, but still using the JC2.2.2 library, to see if that helps. I seem to remember it didn’t work properly with the JC3.0.5 library (though I may revisit this), but surprisingly it compiled and ran with the older library. (The card was sold as 3.0.4 for whatever reason, perhaps this is why)

But it throws an error when the ECConfig() is initialised.