Analyzing Low Level Issues relating to RFID/NFC Cards

Vaarnan Drolia, Eugene Huang,
Poornima Muthukumar, Saberi Sarkar

National University of Singapore
School of Computing
Singapore, 117417
vd@nus.edu.sg, eugenehuang@nus.edu.sg,
poornima@nus.edu.sg, A0074369U@nus.edu.sg

Abstract. This paper describes our efforts in uncovering the security details of the MiFare 1K Smartcard and the EZ-Link CEPAS Smartcard and making attempts to exploit the security flaws to be able to clone the card and impersonate another person. Also, we analysed the EZ-Link Java applet to understand how it validates card transactions and speculated possible ways that this can be exploited to recharge cards as well as read their data relating to travel records and transactions.

1 Introduction

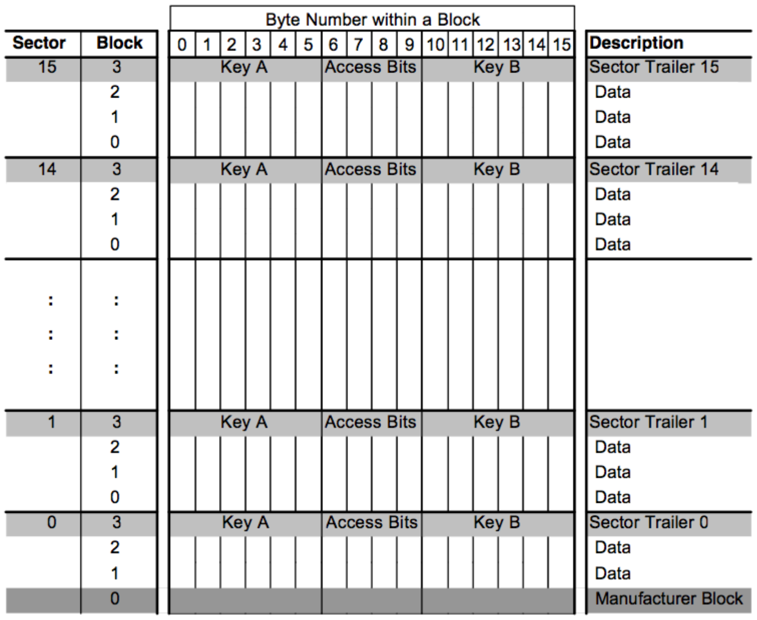
This paper will be divided into two separate parts to deal with the two different types of cards. The first part deals with exploiting the security flaws in the MiFare 1K Smartcard which has several papers published on it. The second part deals with exploring the EZ-Link CEPAS card which is currently deemed to be secure.

2 MiFare 1K Smartcard

The MiFare card is an implementation of the Crypto-1 Cipher which was successfully cracked by Henryk Plotz, a German researcher, and Karsten Nohl, a doctoral candidate in computer science at the University of Virginia.

2.1 Overview

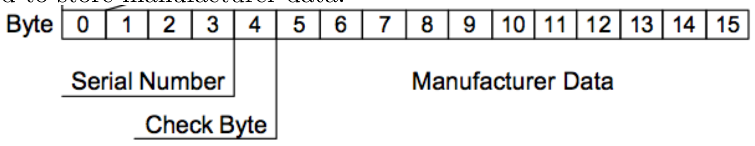
The Mifare 1K is divided into 16 sectors. In each sectors, it consist of 3 blocks and a sector trailer (with the exception of sector 0 having only 2 blocks as block 0 is the manufacturer block). Each block can consist of 16 bytes of data which can be used to store details about the card user. It is also possible to configure a block as a value block but it is not done for the cards mentioned. Hence, we will not go into details about this method.



		Byte Number within a Block																
Sector	**Block**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**Description**
15	3	Key A						Access Bits				Key B						Sector Trailer 15
	2																	Data
	1																	Data
	0																	Data
14	3	Key A						Access Bits				Key B						Sector Trailer 14
	2																	Data
	1																	Data
	0																	Data
:	:																	
:	:																	
:	:																	
1	3	Key A						Access Bits				Key B						Sector Trailer 1
	2																	Data
	1																	Data
	0																	Data
0	3	Key A						Access Bits				Key B						Sector Trailer 0
	2																	Data
	1																	Data
	0																	Manufacturer Block

2.2 Manufacturer Block

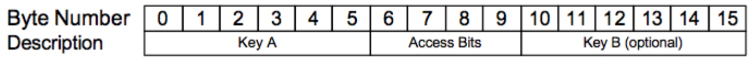
The manufacturer block consist of 3 main components. The first four bytes consist of the serial number (otherwise known as the UID of the card) followed by the check byte, which is calculated by XOR-ing the UID. The last 11 bytes is used to store manufacturer data.



a

2.3 Sector Trailer

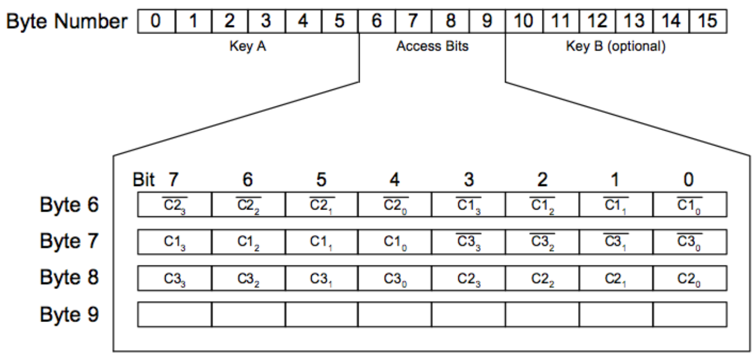
The sector trailer(block 4) for each sector will used to store the the secret key A and B for accessing the sector. Byte 6-9 will store the access bits for accessing the four blocks in each sector.



Byte Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Description	Key A						Access Bits				Key B (optional)					

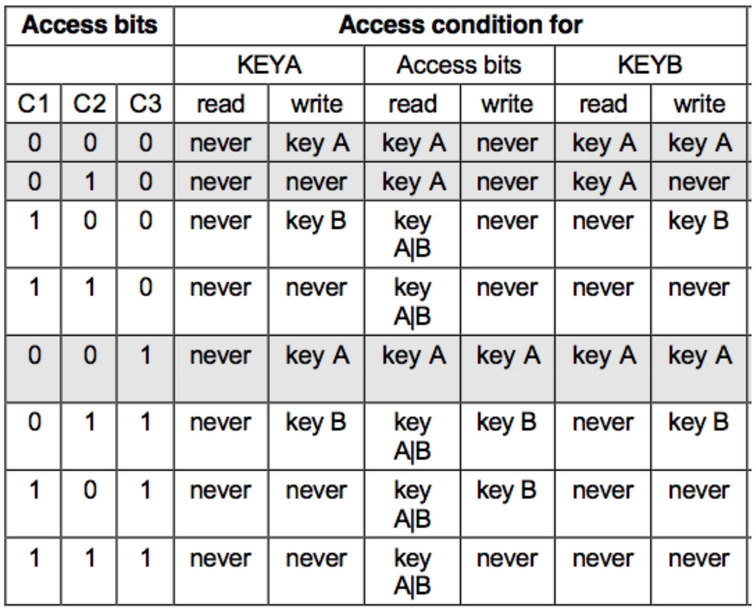
2.4 Breakdown of access bits

The access condition for the data block and sector trailer are stored in the access bit of each sector trailer. For the bit value in CXy, X represents the position of the bit and y represents the block it is used to control access. For example, in order to know the access condition for block 2, we will obtain C12 C22 C32 to compute a 3-bit long representation. We can then refer to the table (data block since is block 2) below to compute the access condition.



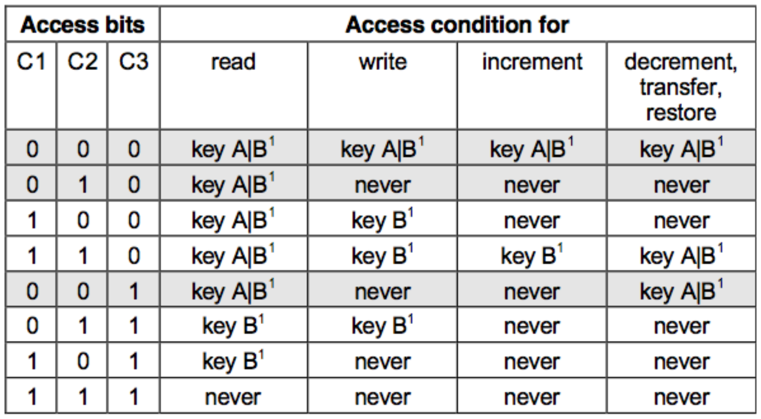
	Bit 7	6	5	4	3	2	1	0
Byte 6	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Byte 7	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Byte 8	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$
Byte 9								

Access condition for data blocks for each sector.



Access bits			Access condition for					
			KEYA		Access bits		KEYB	
C1	C2	C3	read	write	read	write	read	write
0	0	0	never	key A	key A	never	key A	key A
0	1	0	never	never	key A	never	key A	never
1	0	0	never	key B	key A|B	never	never	key B
1	1	0	never	never	key A|B	never	never	never
0	0	1	never	key A	key A	key A	key A	key A
0	1	1	never	key B	key A|B	key B	never	key B
1	0	1	never	never	key A|B	key B	never	never
1	1	1	never	never	key A|B	never	never	never

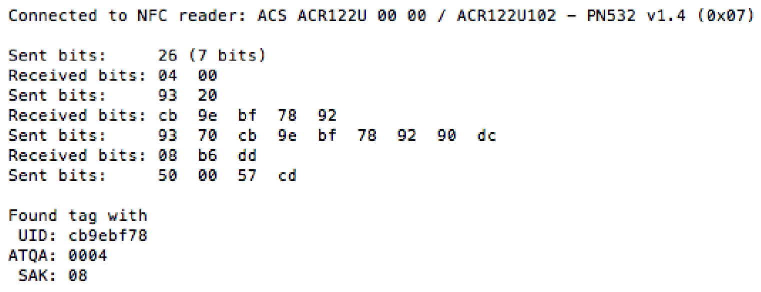
Access condition for sector trailer for each sector.



Access bits			Access condition for			
C1	C2	C3	read	write	increment	decrement, transfer, restore
0	0	0	key A|B[1]	key A|B[1]	key A|B[1]	key A|B[1]
0	1	0	key A|B[1]	never	never	never
1	0	0	key A|B[1]	key B[1]	never	never
1	1	0	key A|B[1]	key B[1]	key B[1]	key A|B[1]
0	0	1	key A|B[1]	never	never	key A|B[1]
0	1	1	key B[1]	key B[1]	never	never
1	0	1	key B[1]	never	never	never
1	1	1	never	never	never	never

2.5 NUS Card

We perform an nfc-anticol on the NUS card. From the results, we found out that the NUS card is a MIFARE 1K card. We then proceed to try to obtain the card dumps for our nus card. We will share our findings below.



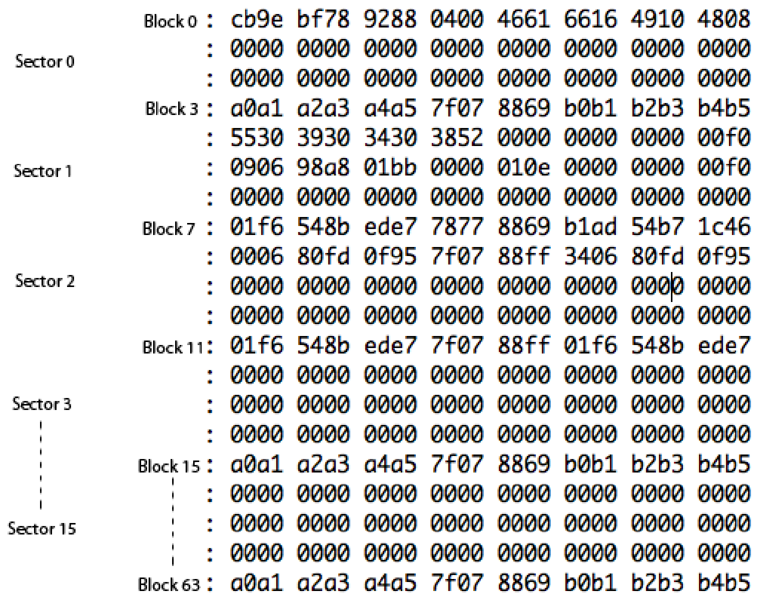
```
Connected to NFC reader: ACS ACR122U 00 00 / ACR122U102 - PN532 v1.4 (0x07)

Sent bits:     26 (7 bits)
Received bits: 04  00
Sent bits:     93  20
Received bits: cb  9e  bf  78  92
Sent bits:     93  70  cb  9e  bf  78  92  90  dc
Received bits: 08  b6  dd
Sent bits:     50  00  57  cd

Found tag with
 UID: cb9ebf78
ATQA: 0004
 SAK: 08
```

As expected, the card dump produced 64 blocks of data. Out of the 16 sectors, we noticed that 14 sectors are actually using known MIFARE default keys. (Only sector-2 and 3 are using unknown keys). Besides this, the data from sector 3 to sector 15 are the same. Therefore, it can be implied that the sectors are not in used.

Below is a sample printout of the dump, sector 4 to sector 14 are left out as they have the same data as sector 3 and 15. We will first attempt to guess the data stored in the various blocks by computing the access conditions for sector 1 and 2.



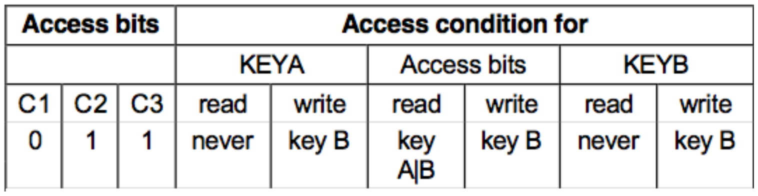
```
           Block 0 : cb9e bf78 9288 0400 4661 6616 4910 4808
                   : 0000 0000 0000 0000 0000 0000 0000 0000
Sector 0           : 0000 0000 0000 0000 0000 0000 0000 0000
           Block 3 : a0a1 a2a3 a4a5 7f07 8869 b0b1 b2b3 b4b5
                   : 5530 3930 3430 3852 0000 0000 0000 00f0
Sector 1           : 0906 98a8 01bb 0000 010e 0000 0000 00f0
                   : 0000 0000 0000 0000 0000 0000 0000 0000
           Block 7 : 01f6 548b ede7 7877 8869 b1ad 54b7 1c46
                   : 0006 80fd 0f95 7f07 88ff 3406 80fd 0f95
Sector 2           : 0000 0000 0000 0000 0000 0000 0000 0000
                   : 0000 0000 0000 0000 0000 0000 0000 0000
           Block 11: 01f6 548b ede7 7f07 88ff 01f6 548b ede7
                   : 0000 0000 0000 0000 0000 0000 0000 0000
Sector 3           : 0000 0000 0000 0000 0000 0000 0000 0000
                   : 0000 0000 0000 0000 0000 0000 0000 0000
           Block 15: a0a1 a2a3 a4a5 7f07 8869 b0b1 b2b3 b4b5
                   : 0000 0000 0000 0000 0000 0000 0000 0000
Sector 15          : 0000 0000 0000 0000 0000 0000 0000 0000
                   : 0000 0000 0000 0000 0000 0000 0000 0000
           Block 63: a0a1 a2a3 a4a5 7f07 8869 b0b1 b2b3 b4b5
```

2.6 Access condition for sector 1

Given 0x7877 8869 for byte 6-9 of block 7 (sector 1 trailer), we can compute the access bit for the data blocks and sector trailer. We first decomposed the access bytes into its bits form. We will ignore byte 9 as it is not used to compute the access bits.

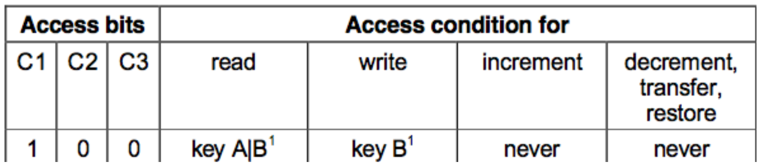
Binary(0x78)	0	1	1	1	1	0	0	0
Access bit	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Binary(0x77)	0	1	1	1	0	1	1	1
Access bit	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Binary(0x88)	1	0	0	0	1	0	0	0
Access bit	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

We compute access bit for sector 1 trailer to be 0 1 1 and from the table, we obtained the following access condition.



Access bits			Access condition for					
			KEYA		Access bits		KEYB	
C1	C2	C3	read	write	read	write	read	write
0	1	1	never	key B	key A|B	key B	never	key B

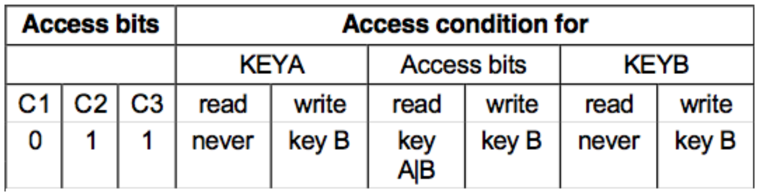
We compute access bit for sector 1 data blocks to be 1 0 0 for all the blocks (4-6) and from the table, we obtained the following access condition.



Access bits			Access condition for			
C1	C2	C3	read	write	increment	decrement, transfer, restore
1	0	0	key A|B[1]	key B[1]	never	never

2.7 Access condition for sector 2

Given 0x7f07 88ff for byte 6-9 of block 11(sector 2 trailer), we can compute the access bit for the data blocks and sector trailer. We first decomposed the access bytes into its bits form. We will ignore byte 9 as it is not used to compute the access bits.

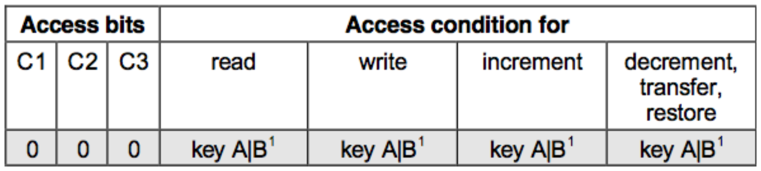


Access bits			Access condition for					
			KEYA		Access bits		KEYB	
C1	C2	C3	read	write	read	write	read	write
0	1	1	never	key B	key A|B	key B	never	key B

We compute access bit for sector 2 trailer to be 0 1 1 and from the table, we obtained the following access condition.

Binary(0x7F)	0	1	1	1	1	1	1	1
Access bit	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
Binary(0x07)	0	0	0	0	0	1	1	1
Access bit	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
Binary(0x88)	1	0	0	0	1	0	0	0
Access bit	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

We compute access bit for sector 2 data blocks to be 0 0 0 for all the blocks (8-10) and from the table, we obtained the following access condition.



Access bits			Access condition for			
C1	C2	C3	read	write	increment	decrement, transfer, restore
0	0	0	key A|B[1]	key A|B[1]	key A|B[1]	key A|B[1]

From the access conditions, we can derived that sector 1 stores more important information than sector 2 as only key B is allowed to write to the data blocks in sector 1.

2.8 Analysis on each block

From all the card dumps we obtained, we concluded that only block 0, block 4, block 5, block 7, block 8 and block 11 were the possible blocks used by NUS.

We will attempt to decode the data and provide an explanation for each block of data.

Matriculation number (Block 4) The matriculation number of the student is stored in block 4 in sector 1. It can be obtained by reading the first 9 byte of the block to have each byte converted from hex to ascii. The sample dump above with 5530 3930 3430 3852 00 will produce U090408R with byte 9 ignored since it is 0.

Student details (Block 5) Out of 16 bytes in block 5, only byte 0-2, 4-5 and 9 appear to store data. Although byte 3 and 8 also have non-zero data, it is the same value for all the card dumps we have obtained. Therefore, we believe it is a false data. Possible values for byte 4-5 are 0x01bb for computing faculty, 0x0052 for engineering faculty and 0x0142 for Science faculty (probably). For the bytes 0-2, we found that the bytes are actually the 6-digit security pin of the card and thus, this is a major security loophole as well as a nice breakthrough. Also, this pin is (to our knowledge) not changeable and normally obtained by going online and keying our card number.

State details (Block 8) Byte 0 is always 0x00 and the value stored in byte 1-5 are always the same as the value stored in byte 11-15. It is unknown what byte 10 of this block is used to represent. However, given the access condition for the sector, we guessed that this block are used to store temporary state for the card or maybe data related to card validity and expiry.

2.9 Cloning our NUS Card

After obtaining our card dumps, we first attempted to over-write one of our existing NUS Cards. This lead to it being rendered unreadable and it showed up as expired on the NUS system. So we procured some blank cards from a local provider and tried to write our card dump into the blank cards. We managed to successfully write the data and tried to test out the card in the schools system. However, we were unable to gain access to the school compounds with the cloned card. We attribute our failure to two possible scenarios.

UID of the card is pre-stored in the NUS system - On tapping of our cards on the reader, the system will obtain the UID from the NUS and verify that the UID is a valid one. If valid, it will then proceed to obtain the details from the card and decides whether to grant access by checking the details against its own access control list.

UID of the card is computed with a secret algorithm and stored within the card - On tapping of our cards on the reader, it will verify the UID with the secret algorithm and the data values on the card before trusting the content of the card. If the computed value is verified to be correct, then will grant access based on its own access control list. However, we think it is highly unlikely as we could not find any such values on our card dump. (As our card were not collected for configurations every semester, we believed that granting access to a specified lab/room is solely done on the NUS system and will not be stored in the NUS card. The NUS card is only used to identify the identity of the card holder).

2.10 Chinese Clone Cards

By default, all block 0 of MIFARE cards are pre-configured to be read-only by hardware specification. However, there are some cards in the market that are said to be freed from this restriction. We believe that we obtained such cards, our clone cards will then worked freely in the school compound and therefore justifies for a possibility that some malicious users will abuse this vulnerability.

2.11 UTown Card

Additionally, we performed nfc-anticol on the UTown card and obtained information that it is also a MiFare 1K Card. To get the dump for the card, it only took us a few seconds as we realised later that all the sectors in the card used the default key 0xFFFFFFFF. Thus, they can also be replicated easily if the Block 0 Chinese Cards are available.

2.12 Conclusion

There are a number of security flaws of the MiFare card that justifies that it should not be used at all. However, by taking advantage of block 0 being read-only, it allows the NUS system to emulate an additional layer of security. As we are unable to obtain a Chinese clone card to try out, we are unable to conclude the possibility of a security flaw in the NUS system. However, if there is indeed such flaw, NUS could also utilize the other sectors on the card to store more data such that it cripples hacker from just cloning the card. A secret algorithm discussed above would provide this additional barrier.

3 EZ-Link Card

The EZ-Link card is a CEPAS (Contactless E-Purse Specification) Smartcard with the commands in this standard following the convention in ISO/IEC 7816-4: 2005. It has Triple-DES security encryption. The main commands are "Read Purse", "Debit Purse" and "Credit Purse". The design allows for partial refund and is limited to the most recent amount debited. The "AutoLoad" and "Cumulative Debit" features are also available. In this standard, key management for controlling e-purse operations is flexible and the final details are decided by the card issuer. Atomicity is a key focus in this standard to ensure reliability of transactions across multiple interfaces.

3.1 Overview

The actual card failed to give us much information of the type and commands supported since it follows the Answer To Request Type B (ATQB) specification and we could not get a response from nfc-anticol.

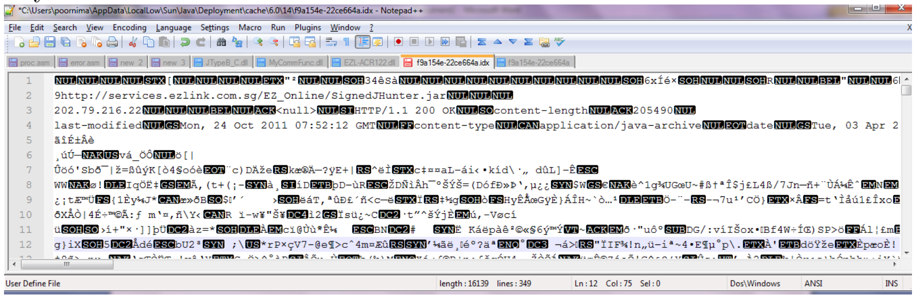
Thus, we chose to instead focus on the Java Applet used by EZ-Link for online top-up of cards and decompile it to try and understand the mechanism of the card and to uncover and weaknesses/secret keys which may be stored in the code.

3.2 Applet Cache

A website which has a Java Applet is downloaded into the users system cache when the user visits the website for the first time. So when the user visits the same website again, the web browser loads the applet from the system cache where it stays until it gets overwritten to make room for other applets as the browser has no knowledge whether an applet file might be needed in the future or not [Oracle, (1993)].Thus, a cache directory is a temporary folder to save Java files mainly to improve performance. Having understood this concept of Applet caching, we looked for the EZ-Link applets cache in the following Java cache directory.

```
C:\Users\poornima\AppData\LocalLow\Sun\Java\Deployment\cache\6.0\14
```

This particular directory had approximately 63 cache folders. When the applet was running we tried to delete the entire cache folder. However the cache folder concerning the EZ-Link applet was not getting deleted because it was currently being used by the applet. This way we discovered the cache folder which belonged to EZ-Link applet. By exploring the cache folder we found that it had two files namely f9a154e-22ce664a and f9a154e-22ce664a.idx.



```
*C:\Users\poornima\AppData\LocalLow\Sun\Java\Deployment\cache\6.0\14\f9a154e-22ce664a.idx - Notepad++

NULNULNULNULSTX[NULNULNULNULETX"²NULNULSOH34êSàNULNULNULNULNULNULNULNULNULNULNULNULNULSOH6xÍé×SOHNULNULSOHRNULNULBEL"NULNUL6
9http://services.ezlink.com.sg/EZ_Online/SignedJHunter.jarNULNULNUL
202.79.216.22NULNULNULBELNULACK<null>NULSIHTTP/1.1 200 OKNULSOcontent-lengthNULACK205490NUL
last-modifiedNULGSMon, 24 Oct 2011 07:52:12 GMTNULFFcontent-typeNULCANapplication/java-archiveNULEOTdateNULGSTue, 03 Apr 2
ãîÉ±Âè
¸úÚ—NAKUSvá_ÖÔNULö[|
Ûöó'Sbð¯|ž=ßûýK[ò4§oóèEOT¨c)DÄžeRSkæ®Ä—?ÿE+|RS^ëÌSTXc‡¤¤aL–ái‹•kíd\·„ dûL]–ÊESC
WWNAKø!DLEIqÖË‡GSEMÃ,(t+(¡–SYNà¸SIíDETBþD–ùRESCŽDÑìÂh¯°ŠÝŠ=(DófÐ»Þ‘,µ¿¿SYN$WGS€NAKè^1g¾UGœU~#ß†ªÎ$j£L4ß/7Jn—ñ+¨ÙÁ¼Ê^EMNEM
¿¡tÆ™ÜFS{1Èy¼J*CANæ»ðBSO$‘’´    ›SOHëáT,ªûÐ£´ñ<c—ëSTXÏRS‡½gSOHòFSHyÊÅœGÿÈ}ÁÎH~`ò…¹DLEETBÖ–¨–RS–¬7u¹’CÖ}ETX×ÂFS=t‘Ìåú1£ÎxoE
ðXÅÒ|4É÷™©Ã:ƒ m‘¤,ñ\Y‹CANR ï–w¥"Š¥DC4ì2GSÏsü¿~CDC2·t”^šÝjÈEMú,–Vøcí
üSOHSO›í+"×·]]þÜDC2àz=*SOHDLEÀEMcï@Ùù*Ê¼  ESCBNDC2#  SYNË KáëpàêªÂ©«§6ý™ÝVT~ACKEMõ·"uô°SUBDG/:víIŠox•Bf4W÷ÍŒ)SP>öFFÁl¦£mE
g}iXSOH5DC2ÅdéESCbU2³SYN ;\US*rP×çV7–@e¶>c^4m¤ÆûRSSYN’¼ãë¸é°?äªENQ°DC3 ¬á>RS"ÏIF¾!n„ü–iª~4•E¶µ°p\.ETXÀ'ETBdöŸžeETXÈpæoÈ!

User Define File    length : 16139    lines : 349    Ln : 12    Col : 75    Sel : 0    Dos\Windows    ANSI    INS
```

3.3 IDX File Information

Valuable Information in the IDX File[Harrell, (2012)]:

- ⋆ Filename: SignedJHunter.jar
- ⋆ URL : http://services.ezlink.com.sg/EZ_Online/SignedJHunter.jar
- ⋆ File size: not sure

⋆ File's last modified date: Mon, 24 Oct 2011 07:52:12 GMT
⋆ IP address where file came from: 202.79.216.22
⋆ Web software involved: Apache
⋆ Deploy request content type: application/ java archive
⋆ Date when file was downloaded: Tue, 03 Apr 2012 15:20:24 GMT

We used the URL link to download the JSignedHunter.jar package. We then used the jad software to decompile and extract the source code.

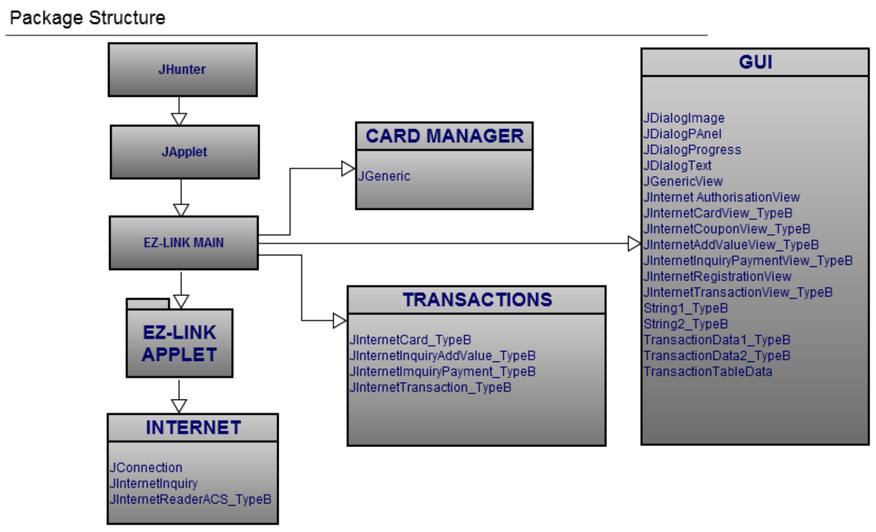
3.4 Architecture and Diagram

The package diagram shows how the various classes are grouped into packages.

The Class JHunter calls JApplet which in turn calls EZ-LINK Main.

EZ-LINK Main depends on the package EZ-LINK APPLET in turn. It contains the class JGeneric which helps the applet to retrieve information from the card. It also contains sub packages like Transaction and GUITransaction. These packages contain the classes which take care of each transaction the customer initiates. GUI contains all the classes responsible for whatever we see on the screen.

Finally the package INTERNET contains the classes responsible for helping the Applet interact with the Internet.



3.5 Java Native Interface

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call, and to be called by, native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.[Wikipedia, (2012)]. This interface is used by the JHunter Applet to load certain DLL files which are stored on the user's computer.

3.6 Javah Utility

The Javah utility is used to generate the C header file from the class file which invokes the native methods.The utility automatically creates the C header file. This header file is then moved to a C++ development evironment (such as Visual Basic) to implement the native methods. Once the implementation is done the C++ code is converted to a DLL(Dynamic Link Library) file to interface with the Java Class[Small, (1998)].

3.7 Sequence of Applet Initialization

1. Applet Class is loaded
2. DLL File is downloaded to the Clients Computer
3. Native method wrapper class is downloaded to the clients computer. (Main class)
4. Wrapper class is loaded
5. Native DLL file is loaded
6. Native Methods are invoked

However, we found that in our EZ-Link Applet the DLL files were saved in the clients computer when the EZ-Link Online Software was installed. While the Main class (also the wrapper class) which invokes the native methods is downloaded in the sandbox by the Applet class when the web site is loaded.

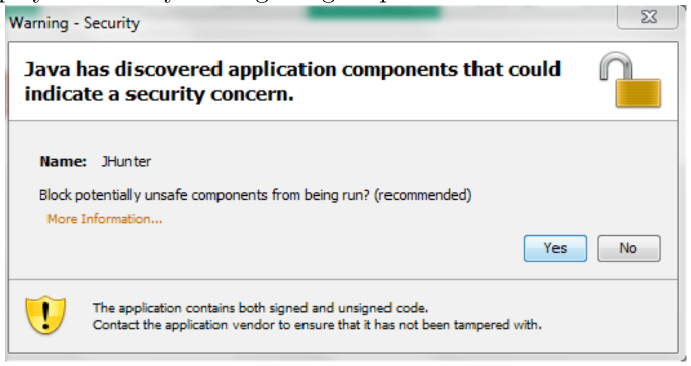
Once the main class is loaded it explicitly loads the DLL files.

```
System.load("C:\\EZLINK\\RWA\\MyCommFunc.dll");
System.load("C:\\EZLINK\\RWA\\EZL-ACR122.dll");
System.load("C:\\EZLINK\\RWA\\JTypeB_C.dll");
```

For the applet to load the DLL files it has to venture out of the sandbox because the DLL files are stored in the following directory which is not the same as the sandbox directory

```
C:\EZLINK\RWA
```

As a result whenever the website is visited, and whenever the applet loads it displays a security message to gain permission from the user.



Warning - Security

Java has discovered application components that could indicate a security concern.

Name: JHunter

Block potentially unsafe components from being run? (recommended)

More Information...

The application contains both signed and unsigned code.
Contact the application vendor to ensure that it has not been tampered with.

3.8 Native Methods

Once the necessary files are downloaded the native method can be used. Every time the native method is used an instance of the wrapper class is created. Main.NativeCloseReaderWriter(); Main.NativeReconnectReaderWriter();

```
public native boolean NativeInitInstance();
public native boolean NativeInitializeACR122();
public native boolean NativeSetRetryTimeToOne();
public native byte NativePollingTags();
public native int NativeRequestResponse(byte byte0, byte byte1,
      int i, byte abyte0[], byte abyte1[]);
public native boolean NativeOpenReaderWriterAuto();
public native boolean NativeReaderWriterIsAlive();
public native boolean NativeReaderWriterIsOpen();
public native boolean NativeCloseReaderWriter();
public native boolean NativeReconnectReaderWriter();
```

As can be observed from the function prototypes, the native methods handle the communication between the reader and the card and communicate these results to the applet.

3.9 Possibilities

APDU In the context of smart cards, an application protocol data unit (APDU) is the communication unit between a smart card reader and a smart card. The structure of the APDU is defined by ISO/IEC 7816-4 Organization, security and commands for interchange. There are two categories of APDUs: command APDUs and response APDUs. A command APDU is sent by the reader to the card it contains a mandatory 4-byte header (CLA, INS, P1, P2) and from 0 to 255 bytes of data. A response APDU is sent by the card to the reader it contains a mandatory 2-byte status word and from 0 to 256 bytes of data [Wikipedia, (2012)].

At the end of the JGeneric.jad file, we found a large number of codes which seemed to be APDU's to communicate with the card. However, we could not communicate with them using the software tried by us. The software used was JSmartCardExplorer at http://www.primianotucci.com/.

We feel, that if one can sniff the communication or obtain some sample communication, the commands can then be constructed using our listing of APDU codes. It might be possible to do this using the Proxmark III Reader.

Checksum We discovered that the card implements a trivial form of check sum calculation where the last digit indicated the checksum.

To calculate the CheckSum each digit of the Card Number is converted from character to integer. Then value is calculated by multiplying each digit with circular sequence 2,9,8,7,6,5,4,3. Following this we calculate the remainder by dividing the value obtained by 11. We check if the remainder is 10 and set it to 0 if it is so.Finally, we then compare the last digit of the card number with the remainder. If both are equal the card number is a valid card number.

Physical ID From the source code we found that the Card Number is always 16 characters long. Out of 16 characters the first eight characters stand for the physical id. For a valid card the Physical Id has to be one of the following. We confirmed this with our cards.

```
"100008002"  "100930007"  "100940006"  "100950005"
"100960004"  "100970003"  "100009000"
```

Card Cloning With the use of the Checksum and Physical ID parts, it might be possible to generate dummy cards in the system which can be passed of as valid cards. The possibility and scope of such an attack is unknown to us.

Payment Server Also, in the JInternetInquiry.jad, we found numbers which we believe are IP addresses but on trying one of the addresses, we did not get any response. Thus, we are not exactly sure about how to use it but it might indicate the servers used for payments. It might be possible to route them to a different server and emulate the recharge of the card.

DLL Injection Since the Java Applet uses the DLL files provided locally by the computer, it may be possible to inject code into them which might make the applet believe it is communicating with a valid card. We attempted to decompile the DLL's using the tool, Boomerang and REC Decompiler, but could only obtain Assembly code.

3.10 Conclusion

We found that the CEPAS card is more obscure in its security since there haven't been any documented breakthroughs on its security being compromised. However, there are still possibilties that can be explored and the online top-up system is bound to have certain weaknesses which can be used to change the card's value.

4 Further Research

Smartcards are pervading into our daily lives and we are moving towards a society where cash is becoming obsolete and confidentiality, integrity and authentication are now being provided by these microscopic hardware devices. There is also an interesting new concept where mobiles will function as cards and will be able to make contactless payments, eg. Google Wallet. In this light, it is becoming increasingly important for consumers to be aware of the risks which come with the adoption of such technologies and more so, for companies which design these technologies to make them secure. The use of open technologies instead of security through obscurity might possibly lead to a greater degree of security.

As far as NUS is concerned, the cards can only do so much damage if cloned, but nonetheless, it is more advisable for us to begin the process of phasing out older, less secure technologies and moving towards better ones. However, the new UTown card is quite the opposite of this philosophy as it is even more insecure and compromising it can give one access to an individuals room. We recommend that NUS implements a well-defined plan to address these security gaps and upgrade from archaic technologies like the MiFare 1K Classic.

5 Acknowledgments

We thank Dr. Hugh Anderson for his help by providing us the card readers and other knowledge required to start our project.

6 References

1. Standard Card IC MF1 IC S50, Philips semiconductor, http://www.itworksolutions.com/brochure/catalogue/RFID/Mifare
2. Images extracted from: Practical Attacks on the MIFARE Classic, Wee Hon Tan, Imperial College London, Department of Computing, http://www.doc.ic.ac.uk/ mgv98/MIFARE_files/report.pdf
3. Harrell, Corey. "Journey Into Incident Response. N.p., 7 Feb. 2011. Web. 5 Apr. 2012. http://journeyintoir.blogspot.com/2011/02/almost-cooked-up-some-java.html.
4. "Java Native Interface"Wikipedia, the free encyclopedia. N.p., n.d. Web. 8 Apr. 2012. http://en.wikipedia.org/wiki/Java_Native_Interface.
5. Small, Steve. "Escape the sandbox: Access native methods from an applet - JavaWorld."Welcome to JavaWorld. N.p., n.d. Web. 8 Apr. 2012. http://www.javaworld.com/javaworld/jw-10-1998/jw-10-apptowin32.html?page=1.
6. "Smart card application protocol data unit" Wikipedia, the free encyclopedia. N.p., n.d. Web. 9 Apr. 2012. http://en.wikipedia.org/wiki/Smart_card_application_protocol_data_unit.
7. "Applet Caching."Oracle Documentation. N.p., n.d. Web. 9 Apr. 2012. http://docs.oracle.com/javase/1.5.0/docs/guide/deployment/deployment-guide/applet-caching.html.