Eavesdropping on and Emulating MIFARE Ultralight and Classic Cards Using Software Defined Radio

Ilias Giechaskiel[1]

Abstract
In this report, we describe a Software-Defined Radio (SDR) approach for eavesdropping on Near Field Communications (NFC) and Radio Frequency Identification (RFID) cards operating at 13.56 MHz. We show that GNU Radio and Python make a great platform for prototyping, while maintaining sufficient performance for passive attacks without extensive optimizations and using only modest processing power. We successfully eavesdrop on real MIFARE Ultralight and Classic 1K cards by capturing the raw radio waves with a home-made antenna. We recover the plaintext of both reader and tag fully by demodulating the incoming radio waves, parsing individual bits and error detection codes into packets, and then decrypting them when necessary. On the transmission side, we achieve full software emulation of the reader and of MIFARE Ultralight and Classic 1K cards (including encryption), and partial hardware emulation, where we correctly modulate the signal, but not within the strict timing limits of the protocol. Our transmissions can also be used to prevent legitimate communication by interfering with the intended reader or tag signals.

[1] *CDT in Cyber Security, University of Oxford, Oxford, United Kingdom*

1. Introduction

Contactless cards and tags have become very popular in recent years, with everyday applications including e-passports [34], ticketing [36, 7, 12], access control [37], and payment [38, 11] systems. However, as these devices operate wirelessly, adversaries can pick up the radio signals and eavesdrop on the communication between a tag and a reader. Traditionally, such attacks on radio communications required dedicated hardware for particular frequencies and modulation types, but with the advent of Software-Defined Radio (SDR), it is possible to use generic equipment and perform the demodulation in software. Even so, despite a range of embedded devices and Field-Programmable Gate Arrays (FPGAs) that are capable of various attacks on Near Field Communication (NFC), Radio Frequency Identification (RFID), and related technologies (Section 2), to the best of our knowledge no open-source SDR implementation exists for High-Frequency (HF) NFC.[1]

To this end, we developed such an implementation on an Ettus Research Universal Software Radio Peripheral (USRP) using Python and GNU Radio with an antenna made out of simple wire that allows passive eavesdropping on reader-tag communication (whose protocols are explained in Section 3). Though our implementation is easily extensible, we focused on MIFARE cards by NXP Semiconductors, since MIFARE has "a market share of more than 77% in the transport ticketing industry", with "150 million reader and 10 billion contactless and dual interface IC's sold" [25]. Specifically, we use Ultralight [29] and Classic 1K [27] cards, as the former does not employ any encryption, while the latter uses a broken cryptographic algorithm (Section 3.4), making them ideal candidates for such exploration. Moreover, we achieve full software and partial hardware reader and tag emulation, that can also be used to jam signals between a legitimate tag and reader. In summary, our contributions (detailed in Section 4 and evaluated in Section 5) are as follows:

1. We implement in pure Software-Defined Radio a demodulator for NFC/RFID readers and tags operating in the 13.56 MHz frequency, which decodes radio waves into plaintext packets.
2. We test our implementation by eavesdropping on real MIFARE Classic 1K and Ultralight communications with an RFID reader using a home-made antenna and a USRP, successfully decoding any encrypted packets.
3. We additionally implement in software the emulation of both readers and tags, including encryption if necessary.
4. Though our transmission capabilities cannot keep up with the NFC timing requirements, we show how our implementation can jam real reader-tag transmissions.
5. Overall, our work shows that prototyping using Software-Defined Radio is sufficient in practice for passive attacks, without the need for extensive optimizations or heavy computing power.

2. Related Work

Early work on RFID Hacking was conducted in a non-academic context, and focused on finding vulnerabilities in access control systems [32, 36]. Later, Buettner and Wetherall [5, 6, 4] experimented more systematically with RFID and SDR, but focused primarily on Gen 2 cards operating at 900 MHz. Their work was extended by others, typically in the context

[1] Though they exist for UHF Gen2 cards. See `https://github.com/brunoprog64/rfid-gen2` and `https://github.com/yqzheng/usrp2reader` for instance.

of proposing better protocols [3, 40], but still for Ultra High Frequencies (UHF), with the exception of a recent work by Hassanieh et al. [14], which also included an extension to HF.

There have also been a number of designs which use microcontrollers and Field-Programmable Gate Arrays (FPGAs) for signal processing, such as the Proxmark 3 [39], and RFIDler [21]. Though such projects allow the use of custom firmware for additional functionality, they also require dedicated hardware in their design.

The MIFARE Classic cryptographic protocol was reverse-engineered by Nohl et al. by dissolving the plastic surrounding the chips, recovering the individual logic gates and converting them to a high-level algorithm [24]. Garcia et al. then discovered additional vulnerabilities of the protocol based on its nested authentication and parity bits [12]. Due to the wide range of applications of the MIFARE Classic, the topic became very popular for Master's thesis projects [33, 7, 31, 37], which found additional vulnerabilities, or examined the problem within the context of a specific application.

Finally, given the widespread availability of NFC-enabled mobile devices, researchers have also focused on NFC relay attacks using mobile phones [38, 11], as well as exploring [23] and protecting [13] the NFC mobile phone stack.

3. Background

The terms Radio Frequency Identification (RFID), Near Field Communication (NFC), contactless smartcards, proximity cards and vicinity cards are often used interchangeably, but they are covered by different standards and concern different parts of the radio spectrum. In this project, we looked at the ISO/IEC 14443 standard, with physical characteristics defined in [15], modulation and encoding in [17], initialization and anticollision in [18] and transmission protocols in [16].

Specifically, we focus on **Type A** communications, whose **carrier frequency** is $f_c = 13.56$ MHz. The reader – or **Proximity Coupling Device (PCD)** – communicates with the card – or **Proximity Integrated Circuit Card (PICC)** – through **100% Amplitude Shift Keying (ASK)**, with data using the **Modified Miller** encoding (Section 3.1). The communication from tag to reader utilizes **Load Modulation** with a **subcarrier frequency** of $f_s = f_c/16 = 847.5$ kHz, using **Manchester** encoding (Section 3.2), and both transmit 106 kbit/s.

Because the carrier frequency is $f_c = 13.56$ MHz, the wavelength is $c/f_c \approx 22$ meters, making it impossible to deploy antennas that would fit in a card-size form-factor. Additionally, because the cards are **passive** (i.e. do not have their own power source), both the communication and the power source are achieved through **inductive coupling** from the PCD's antenna loop to the PICC's antenna loop. We discuss the high-level protocol in Section 3.3, and the MIFARE Classic encryption algorithm in Section 3.4.

3.1 PCD Transmissions

Amplitude Shift Keying (ASK) of depth $X\%$ is a form of digital modulation which specifies that if the amplitude of the signal representing a digital 1 is equal to A and the amplitude of the signal representing a digital 0 is equal to B, then $X = \frac{A-B}{A+B}$, as showing in Figure 1. Specifically, for 100% ASK, no signal is sent at all during a digital 0, indicating that such periods must be very brief, since the PICC needs to keep charge (using a capacitor) for the period of silence.

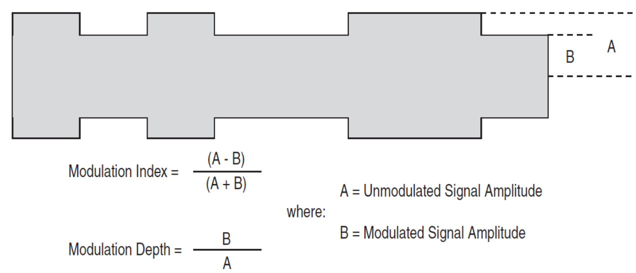

Figure 1. Amplitude Shift Keying (ASK) [1]

This is achieved through the Modified Miller Encoding, which ensures that there is no period of more than $3\mu s$ of silence ("pause"). Specifically, every bit is represented as a (combination of) signals lasting a total of $t_b = 128/f_c \approx 9.44\mu s$. The encoding (show in Figure 2) is as follows:

- A 1 is encoded as an unmodulated signal for $t_b/2 \approx 4.72\mu s$, followed by a period of silence for $3\mu s$, followed by an unmodulated signal for $t_b/2 - 3 \approx 1.72\mu s$
- A 0 after a 0 is encoded as a silence period for $3\mu s$ followed by an unmodulated signal for $t_b - 3 \approx 6.44\mu s$
- A 0 after a 1 is encoded as an unmodulated signal for a period of $t_b \approx 9.44\mu s$
- To indicate the beginning and the end of a transmission, a 0 bit is inserted at both the start and the end

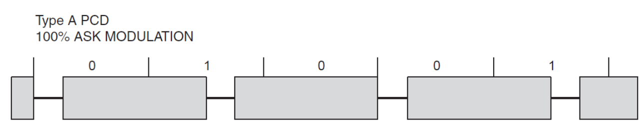

Figure 2. Miller Encoding at 100% ASK [1]

In practice, however, because of hardware imperfections, the pause is not perfect, but needs to comply with the requirements shown in Figure 3. As a result, the modulated carrier for the encodings resembles Figure 4.

3.2 PICC Transmissions

As mentioned above, the tag does not have sufficient power for active transmissions. Consequently, the PICC achieves data transmission passively, by changing its *load*, which can be inferred as a voltage drop on the PCD, hence the term **Load Modulation**. Switching the load generates a **subcarrier**, which has a frequency $f_s = f_c/16 = 847.5$ kHz.

The bits are then encoded using **On-Off Keying (OOK)** or **Manchester Encoding** as follows, with a total duration also equal to $t_b \approx 9.44\mu s$, which also equals 8 periods of the subcarrier, also shown in Figures 5 and 6:

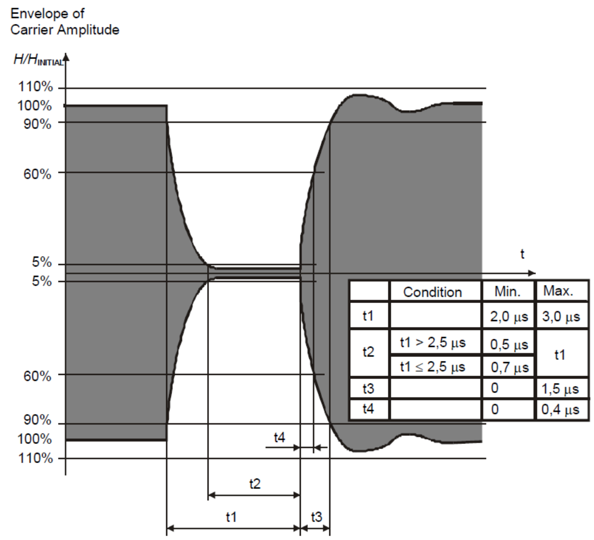

Figure 3. Real Pause Requirements [17]

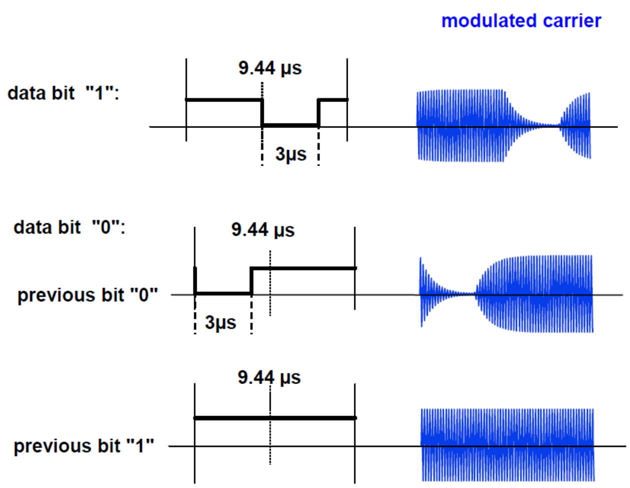

Figure 4. Realistic Miller Encoding [25]

- A 1 is encoded by modulating the subcarrier for the *first* half ($= t_b/2 \approx 4.72\mu s$) of the bit duration
- A 0 is encoded by modulating the subcarrier for the *second* half ($= t_b/2 \approx 4.72\mu s$) of the bit duration
- A logical 1 starts the transmission
- No modulation signifies the end of a transmission

3.3 The Protocol

Though the ISO/IEC 14443A protocol is general, we will focus on a few key aspects that are relevant to our discussion. As a result, we will refer the reader to [18] for more details such as timing requirements.

First of all, it is worth noting that each byte is **ordered** from the Least Significant Bit (LSB) to the Most Significant Bit (MSB), and that each byte is followed by an **odd parity bit**, meaning that an even number of high bits (ones) is followed by another high bit (one), whereas an odd number of ones is followed by a low bit (zero). For example, the byte 0x3F is encoded as 1111 1100 1.

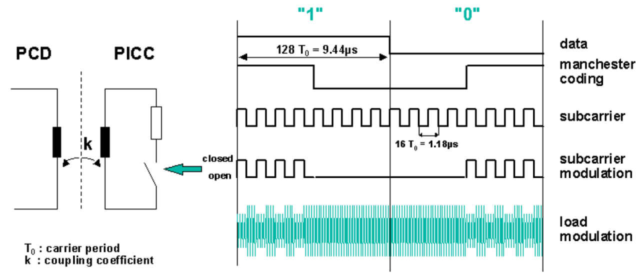

Figure 5. Manchester Encoding with Load Modulation [25]

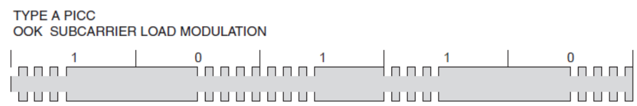

Figure 6. Envelope of Load Modulation [1]

The PCD signifies that it is waiting to read tags by repeatedly sending a **REQA (0x26)** or a **WUPA (0x52)**, where the "A" signifies that type A protocol is used. The difference between the REQA request and the WUPA wake-up request is that the latter also wakes up PICCs that were previously asked to HALT. Unlike all other commands, they are sent using a *short frame* that only consists of 7 bits, and which does not include a parity bit. As a result, the REQA command (including the beginning and end transmission zero bits) is sent as the sequence of bits 0 0110 010 0.

The standard has also defined **Anticollision** and **Selection** phases before the transmission of actual data which are used to ensure the non-interference from multiple tags and the correct selection of the tag. However, we only discuss them in the context of the MIFARE cards in Appendix A, since they are not necessary for understanding the rest of this report.

Finally, we mention that to detect errors with longer transmissions, for some requests and responses a **Cyclic Redundancy Check (CRC)** is used on the transmitted bytes (but excluding start/end and parity bits). The polynomial used is $x^{16} + x^{12} + x^5 + 1$, with a starting value of 0x6363, under the assumption that "FF0 shall be the leftmost flip-flop where data is shifted in [and] FF15 shall be the rightmost flip-flop where data is shifted out" [18]. For example, the **HALT/HLTA** command uses 2 bytes (0x50 0x00) followed by the two CRC bytes which can be calculated as 0x57 0xCD.

3.4 MIFARE Classic 1K Encryption

Even though the MIFARE Ultralight is ISO/IEC 14443 A compliant [29], the MIFARE Classic 1K uses a proprietary cryptographic protocol called CRYPTO1 [27]. The details of the protocol were not made publicly available, with the MIFARE datasheets only broadly explaining the 3-pass protocol [27]. Each **sector** (equal to 4 **blocks** of 16 bytes each) has two 6-byte keys (**Key A and B**), which on delivery are set to

[0xFF 0xFF 0xFF 0xFF 0xFF 0xFF] (but can be changed later on a per-sector basis). Each authentication happens with one of the two keys chosen by the reader, and can only be used to access a specific sector. Each sector contains one block (the **sector trailer**) which contains the two keys and some **access bits** which determine the allowed operations for the 4 blocks (See Appendix A.2 for more details).

The PCD indicates to the PICC that it wants to authenticate through a command indicating which key to be used and what address to use. As shown in Figures 7 and 8, the three-pass scheme consists of a 4 byte challenge sent from the PICC to the PCD (Token RB), an 8-byte challenge and response (of 4 bytes each) send from the PCD to the PICC (Token AB) and a response from the PICC to the PCD (Token BA), if the reader's encryption was correct. After the first challenge (Token RB), **all** traffic is encrypted, even subsequent authentications, which leads to a weakness known as **nested authentication** [12].

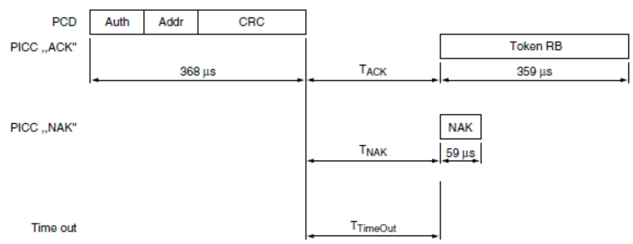

Figure 7. Authentication Part 1 [27]

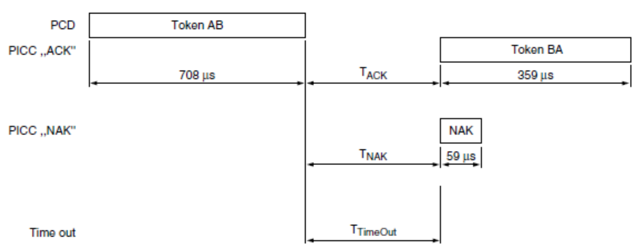

Figure 8. Authentication Part 2 [27]

Though we discuss the encryption algorithm in greater detail in Appendix B, it is worth noting a few things based on the research in [24, 12]. The CRYPTO1 encryption scheme is a stream cipher which consists of a 48-bit (equal to the key length) Linear Feedback Shift Register (LFSR) and a non-linear **filter** function [24]. The encryption incorporates both the tag's Unique Identifier (UID) and the random nonce RB, which however is only generated using a 16-bit LFSR. Nonetheless, both challenge responses only depend on the tag's nonce (and not the reader's nonce or the UID), and use the same LFSR as the Random Number Generator (RNG).

What is more, the parity bits are also encrypted (making the MIFARE Classic 1K *incompatible* with the ISO 14443 protocol), and "the bit of keystream used to encypt the parity bits is reused to encrypt the next bit of plaintext" [12]. This vulnerability, in combination with the nested authentication mentioned above (which causes the token RB to also be encrypted) leaks data which can be used to guess the nonce or to reveal the secret key.

4. Implementation

In this section we discuss our setup and methodology (Section 4.1), the design of the antenna used (Section 4.2), as well as the approach used for decoding transmissions (Section 4.3) and for emulating them (Section 4.4).

4.1 Setup and Methodology

For this project, we used Ettus Research's Universal Software Radio Peripheral (USRP) N210 [10], in combination with the BasicRX/TX and LFRX/TX daughterboards [9], both of which cover the 13.56 MHz frequency. The USRP has become the de-facto SDR platform in combination, and also allows custom code to be written on its FPGA, which we did not pursue in this project. Instead, all signal processing was done on a laptop, using Python and the GNU Radio toolkit/framework, which is easily extensible and provides many building blocks ("modules") that can be incorporated into new designs.

For lack of a better alternative, the laptop used is a Samsung NP900X4C with an Intel i5-3317U @ 1.7 GHz and 8 GB of RAM, but the operating system used (Kali Linux 1.1.0a) was booted off a 16 GB USB 3.0 Lexar JumpDrive. Moreover, due to the lack of an Ethernet port, the USRP was connected to the laptop on a Plugable USB3-E1000 USB 3.0 Gigabit Ethernet Adapter.

To measure signals accurately, we used a Rigol DS2302A digital oscilloscope with two 300 MHz channels [35]. The actual reader was an RFID-RC522 module using the MFRC522 chip by NXP Semiconductors [30] connected to an Arduino UNO using the Serial Peripheral Interface (SPI) and the DumpInfo example at [2]. A schematic is found in Figure 9, while Figure 10 is a picture of the module, where the test pad for the reception (RX) part of the antenna is highlighted.

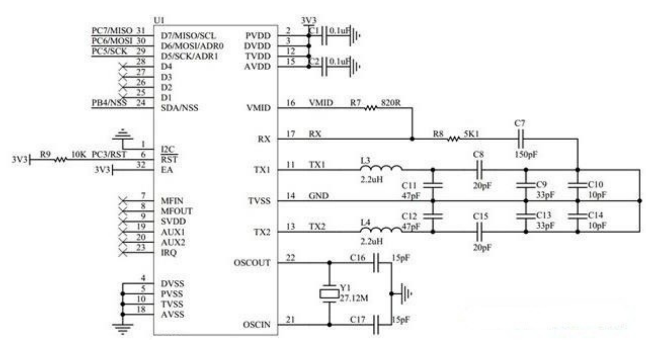

Figure 9. RC522 Schematic `http://img.alibaba.com/img/pb/082/266/824/824266082_485.jpg`

The types of tags/cards used are shown in Figure 11 and are a Classic 1K card, and an NTAG203 [28], which is compatible with the Ultralight. The NTAG203 is actually larger and has a different memory layout, but the Arduino library

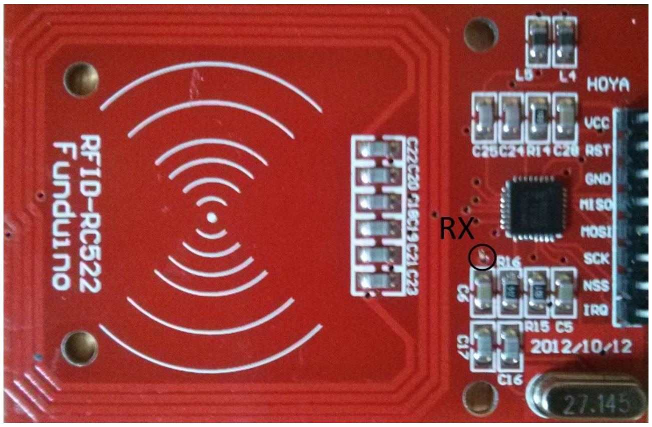

Figure 10. RC522 RX Test Pad

used does not distinguish between the two, so only 16 out of the 42 pages are revealed. It is important to note that although we only two *types* of cards were used, more than one actual card per type was tried with identical results.

Figure 11. Cards/Tags Used in the Experiment

4.2 The Antenna

Much research has been conducted into making RFID-type antennas work well and up to a large distance, with much of it available as application notes [25, 22, 26]. Many of them are also available for direct purchase, such as the DLP-RFID-ANT by DLP Design,[2] but fundamentally the RFID antenna is just an inductor, made out of wire wrapped into a coil. These home-made antennas made out of simple wire (or out of NFC tags themselves [20]) have proven themselves to work [8], so we made our own. According to [26], the inductance should be between 300 nH and 3 μH, so using Equation (1) with $N = 8$ turns, $D = 4$cm, and $s = 2.8$mm (22 AWG), we get an inductance of 5.28μH, which is within the prescribed limits.

$$L[nH] = \frac{24.6 \cdot N^2 \cdot D[cm]}{1 + 2.75 \cdot \frac{s[cm]}{D[cm]}} \tag{1}$$

[2] Found at `http://www.dlpdesign.com/rf/ant1.shtml`

Measuring the signal strength of the wire loop and of the RC522 RX test pad directly on the oscilloscope resulted in equal signal strength (when they were less about a cm apart), but connecting it to the USRP made the signal strength drop considerably, and also required a tuning capacitor in series with the antenna.

For emulating PICC transmission (TX) through Load Modulation, we used a transistor and 2 resistors. Since the entire set-up was crude, their values were empirically determined, but the schematic can be seen in Figure 12 and the final circuit in Figure 13.

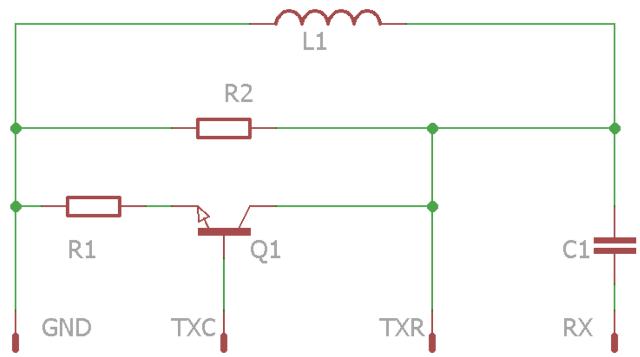

Figure 12. Antenna Schematic

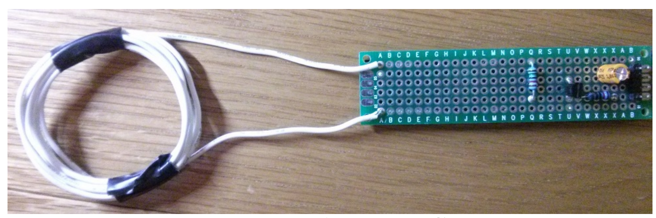

Figure 13. Antenna PCB

Though we discuss the antenna more in Section 5, we mention here that the range for eavesdropping on both reader and tag (which is further away) was only about 1-2 cm. To remedy this, we taped the antenna to the back of the RC522 reader, and the tag to be read to the front of it as shown in Figure 14. The picture taken is before the final PCB was made, and with a different NFC tag (still using the NTAG203 chip).

4.3 Eavesdropping

For consistency/determinism, and easy testing/reproducibility, we initially recorded the interaction between the reader and the tag. This was achieved by using our home-made antenna connected to the USRP in the setup of Figure 14. The *envelope* of the signal is sufficient for our purposes, and for our Amplitude Modulated (AM) signal can be calculated as the absolute value of the signal. Recoding the envelope as a WAV file (using GNU Radio's `wavfile_sink`) using a sample rate of 2,000,000 samples/second and a 16-bit output, we get around 4 MB of data to be processed per second.

Opening the resulting audio file in Audacity Audio Editor, we see that for PCD transmissions the signal drops close to

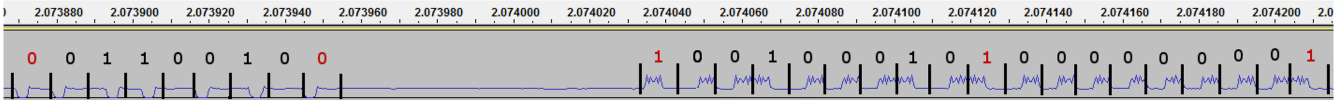

Figure 15. Annotated Recoding

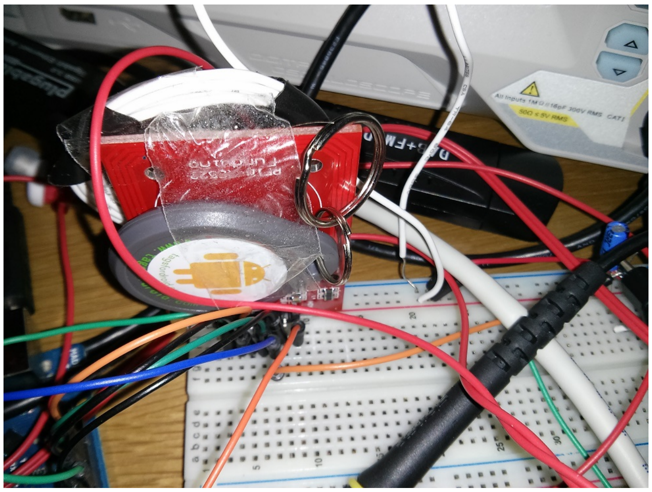

Figure 14. Measurement Setup

0, while for PICC transmissions the subcarrier spikes hover at about 5-10% above the average. An annotated example of the REQA and ATQA transmissions is shown in Figure 15, where start/end and parity bits are shown in red.

Consequently, we can detect such transitions by having a moving window (say of length $2,000$) that keeps track of the average, and if the next value is below the *low* threshold (10% of the average), it is considered part of the reader transmission, and if it goes above the *high* threshold (110% of the average), a part of the card transmission. The transmission is considered over when the signal has returned to its average values for too long (currently $> 25\mu s$), and to ensure that the average does not drift, values above the high threshold and below the low threshold are not included in the moving average. The duration and values of these transitions are then passed on to the appropriate decoders through callbacks, so that the decoding runs in a background thread.

The two decoders (one for Manchester Encoding, and the other for the Modified Miller Encoding) are essentially Finite State Machines (FSM) that implement the specifications mentioned in Sections 3.1 and 3.2, allowing for some margin of transmission and measurement errors. For instance, the sequence $[(0,3),(1,11),(0,3),(1,16),(0,3),(1,6)]$ of PCD (bit, μs duration) pairs would be (approximately) split as:

$$\begin{aligned}
[(0,3),(1,6.5)] &\rightarrow 0 \\
[(1,4.5),(0,3),(1,1.5)] &\rightarrow 1 \\
[(1,9.5)] &\rightarrow 0 \\
[(1,5),(0,3),(1,1.5)] &\rightarrow 1 \\
(1,4.5) &\rightarrow \text{state=ONE_FIRST_STAGE}
\end{aligned}$$

Having recovered all the bits (including parity, but discarding any start/end bits), and knowing whether the PCD or the PICC is doing the transmission, these bits can be interpreted with context in a more high-level FSM. This is the code that updates the internal state of the tag/reader when needed (e.g. to decrypt bits in the MIFARE Classic case), ensures that the parity is correct and transforms bits into bytes, and then based on the current state and the "header" of the incoming bytes determines the command issued, interprets the bytes and checks any CRC if necessary.

For unencrypted commands, this process is not crucial,[3] but for the Classic 1K, parsing the commands to get the UID, for instance, is of utmost importance as any deviations would result in ciphertext that cannot by decrypted. Specifically for regular data transmission decryption is straightforward, but the setup phase of the challenge-response protocol needs to be handled more subtly, especially because of nested authentications after the first authentication. Hence, while the actual cipher is abstracted away into a different class, it is the responsibility of the FSM to correctly call it.

4.4 Emulating

The FSM proved to be an important abstraction for the emulation part of the project because it centralized all encryption considerations. As a result, the emulated Reader and Tag only deal with plaintext messages (and not individual bits). Specifically, the Reader is set to perform identically to the Arduino's `DumpInfo` program, by performing the anticollision, and then reading all card blocks. The Tag is programmed to respond to the incoming commands, and its memory layout is set dynamically through files. For reproducibility (and especially when emulating the tag against a recording), the randomness used by the Tag and the Reader can also be fixed, but it can also be generated on-the-fly as needed, ensuring that this is not merely a replay attack.

Because of this setup, it is possible to emulate both the reader and the tag simultaneously, without needing to go through encoding and modulation, but we have also coded the Manchester and Modified Miller encodings (in a reverse fashion to Section 4.3), as well as modulation, so that they can be output to a WAV file for use without a USRP.

Modulating the Reader's output is straightforward: it is enough to generate a 13.56 MHz (sine wave) carrier and output it, or output nothing for the "pause" duration. Load Modulation of the Tag is more complicated, and is achieved by only outputting when the encoding is a logical 1 (for either $4.5\mu s$ or $9\mu s$). Specifically, however, and as explained in Section 3.2 (Figure 5), this is achieved by generating a 847.5

[3] The code will just print the plaintext bytes for unknown commands.

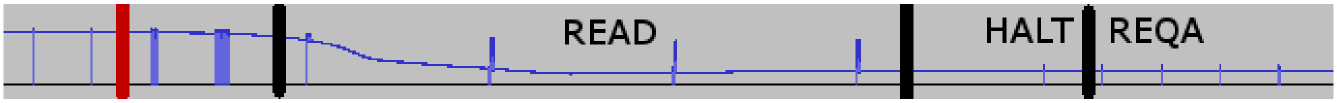

Figure 16. Signal Received when Approaching Tag

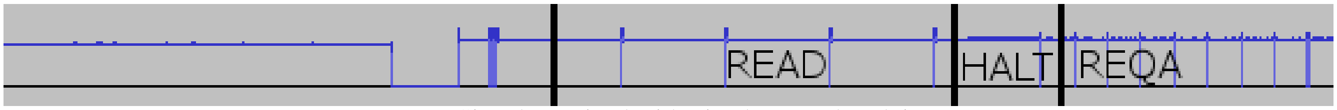

Figure 17. Signal Received with Fixed Tag and Arduino Reset

kHz subcarrier, multiplying by the bit to output, and then switching the load (in this case through the transistor) only for the amount of time for which the signal is positive.

5. Evaluation

In this section, we take a critical look at our approach for eavesdropping (Section 5.1) and for emulating (Section 5.2).

5.1 Eavesdropping

First of all, it is worth mentioning that with recorded reader-tag communications, the eavesdropping code behaves predictably and always correctly decodes the messages. However, at least initially, the code required 50 seconds of processing per 1 second of data. This was due to the fact that incoming messages in GNU Radio Python code are stored as `NumPy` arrays which do not support efficient iteration. Converting them to a list before iteration resulted in a 10× improvement, and assigning local names to function calls resulted in an additional 2.5× improvement, for a processing cost of about 2.2 seconds per 1 second of data. This is still not real time, but given the unusual setup which relies heavily on the USB bus, this performance is acceptable given the convenience of prototyping in Python.

That said, the setup could have substantially benefited from a more fine-tuned antenna. As mentioned in Section 4.2, the range was only a couple of centimeters, and this was certainly not improved by the lack of a proper connecting cable and shielding. In the Do-It-Yourself (DIY) spirit, we also attempted to create an amplifier (which would not be a Low-Noise) based on the design in [19] with a couple of minor resistance modifications to make it work with 9V. However, the modifications did not quite work, but instead resulted in signal distortion as shown in Figure 18.

This setback thus necessitated our setup (Figure 14) where the PCD, the PICC, and our antenna are in close proximity. However, we also tested a more "real-world" situation, where the user approaches the reader with the tag. This is shown in Figure 16, where the signal strength drops when the tag starts approaching the reader (due to their coupling). As can be seen, the signal strength is not sufficient to recover the tag's transmission during the anticollision and selection phases, but it can recover them after the first READ request. This would be insufficient for MIFARE Classic cards, but it does result in partial recovery for the Ultralight card.

It is worth pointing out that the Reader commands are fully decoded even during the signal strength changes. This remains true and is even more pronounced in Figure 17, where the Arduino is reset (but the tag remains close to the reader). The moving average methodology, then, works well, and we have included a couple of example traces decoded in Appendix C for reference and completeness.

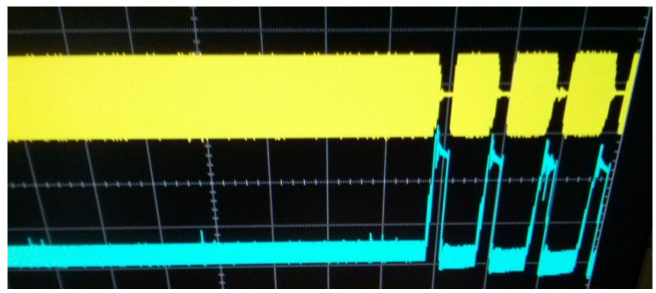

Figure 18. Original in Yellow and "Amplified" in Blue

5.2 Emulating

The software emulation (either of both reader and tag, or of one of them against a recorded WAV file), was proven to work both by checking the plaintext output, and by running the output WAV file of the emulation through the eavesdropping code to recover the transmission. However, the emulation does not adhere to the ISO and MIFARE timing requirements. Specifically, since the signal processing is not real-time, it did not seem prudent to focus on clock recovery/synchronization and implementing timeouts, as it would be impossible to test them within the confines of our setup.

However, this makes it harder to test the transmissions against real hardware. Testing the reader emulation is somewhat easier because the reader generates its own carrier. Specifically, as shown in Figure 19, the emulated signal closely resembles the signal of a real reader (Figure 20). Moreover, as shown in Figure 21, we see in yellow that the USRP signal sent is not particularly strong, and the reception on the (unpowered) RC522 reader antenna in blue already indicates a strength drop due to our untuned transmissions.

The implementation of the hardware tag emulation was not as successful. Specifically, even with the LFTX daughterboard, we could not get any signal transmission out of the USRP at the 847.5 kHz range, so we had to up the frequency to 13.56 MHz. The behavior of the signal strength is similar

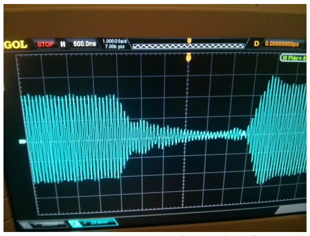

Figure 19. Emulated Signal

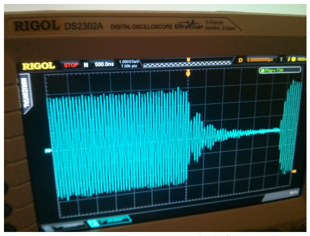

Figure 20. Real Signal

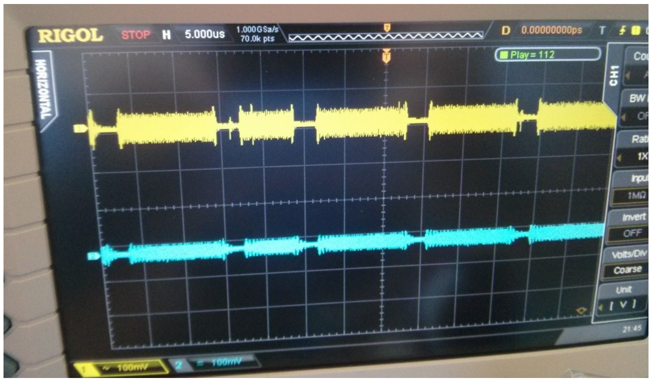

Figure 21. Reader TX in Yellow, RC522 Antenna in Blue

(Figure 22), but it is unclear why the signal is alternating, since it is non-negative before exiting the USRP (Figure 23).

Figure 22. Tag TX in Yellow, RC522 Antenna in Blue

Moreover, because of the lack of synchronization, when the RC522 Reader is on its REQA may overlap with the transmitted signal (Figure 24). However, this can be used to our advantage and jam signals: doing (dynamic) interference in response to data being sent (although with some processing lag) results in timeouts and other errors at the Arduino end.

6. Conclusions and Future Work

All in all, the Software-Defined Radio (SDR) approach for eavesdropping and emulating MIFARE Classic 1K and Ultralight cards proved to be very fruitful for an initial exploration and prototyping phase. That said, the work was hindered by the setup (both in terms of using USB 3.0 and in terms of the antenna), but it proved to be adequate for non-real-time processing and without any extensive optimizations.

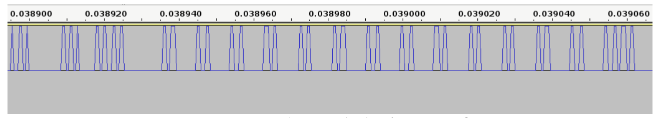

Figure 23. Load Modulation Before TX

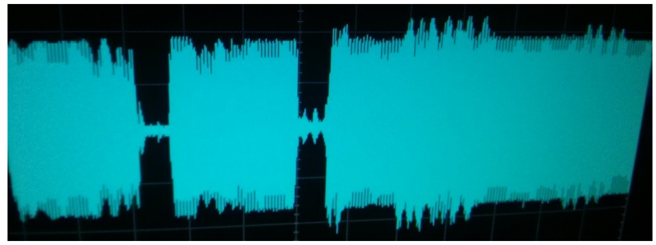

Figure 24. Tag TX Overlap with RC522 Reader

However, it seems that coding in C++ or directly on the FPGA would yield much better results (especially for higher sampling rates), so future work could focus on comparing the SDR approach with embedded platforms. Moreover, looking at newer cards such as the MIFARE Ultralight EV1 and the MIFARE DESFire EV2 would be an interesting extension, especially in the context of testing the platform in the real world, where MIFARE Classic cards have been rendered obsolete. Finally, more focus could be placed on respecting the timing requirements of the protocol — which could be improved by doing clock recovery for synchronization — but we firmly believe that this work is a good start.

Acknowledgments

I would like to thank Kasper Rasmussen for being my supervisor, for his invaluable help throughout the project, and for entrusting me with his expensive equipment. I would also like to thank Simon Crowe for helping shape the focus of the project. Finally, I would like to thank Kellogg College and their Research Support Grant which enabled the purchase of some of the components that were used in this project.

References

[1] Atmel. *Understanding the Requirements of ISO/IEC 14443 for Type B Proximity Contactless Identification Card*, 11 2005.

[2] Balboa, M. Arduino RFID library for MFRC522. `https://github.com/miguelbalboa/rfid`. Acc.: 2015-06-19.

[3] Briand, A., Albert, B., and Gurjao, E. Complete software defined RFID system using GNU radio. In *IEEE International Conference on RFID-Technologies and Applications (RFID-TA 2012)*.

[4] Buettner, M., and Wetherall, D. A software radio-based UHF RFID reader for PHY/MAC experimentation. In *IEEE International Conference on RFID 2011*.

[5] BUETTNER, M., AND WETHERALL, D. A flexible software radio transceiver for UHF RFID experimentation. Tech. Rep. UW-CSE-09-10-02, University of Washington - Computer Science Department, 2009.

[6] BUETTNER, M., AND WETHERALL, D. A "Gen 2" RFID monitor based on the USRP. *SIGCOMM Comput. Commun. Rev.* (2010).

[7] DE KONING GANS, G. Analysis of the MIFARE classic used in the OV-Chipkaart project. Master's thesis, Radboud University Nijmegen, 2008.

[8] DIAKOS, T., BRIFFA, J., BROWN, T., AND WESEMEYER, S. Eavesdropping near field contactless payments: A quantitative analysis. *IET Jour. of Eng.* (2013).

[9] ETTUS RESEARCH. *TX and RX Daughterboards For the USRP Software Radio System*.

[10] ETTUS RESEARCH. *USRP N200/N210 Networked Series*.

[11] FRANCIS, L., HANCKE, G., MAYES, K., AND MARKANTONAKIS, K. *Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones*.

[12] GARCIA, F. D., ROSSUM, P. V., VERDULT, R., AND SCHREUR, R. W. Wirelessly pickpocketing a Mifare Classic card. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (SSP 2009)*.

[13] GUMMESON, J. J., PRIYANTHA, B., GANESAN, D., THRASHER, D., AND ZHANG, P. Engarde: Protecting the mobile phone from malicious nfc interactions. In *11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2013)*.

[14] HASSANIEH, H., WANG, J., KATABI, D., AND KOHNO, T. Securing RFIDs by randomizing the modulation and channel. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*.

[15] ISO. Identification cards—Contactless integrated circuit cards—Proximity cards—Part 1: Physical characteristics. ISO 144443-1:2008, International Organization for Standardization, 2008.

[16] ISO. Identification cards—Contactless integrated circuit cards—Proximity cards—Part 4: Transmission protocol. ISO 144443-4:2008, International Organization for Standardization, 2008.

[17] ISO. Identification cards—Contactless integrated circuit cards—Proximity cards—Part 2: Radio frequency power and signal interface. ISO 144443-2:2010, International Organization for Standardization, 2010.

[18] ISO. Identification cards—Contactless integrated circuit cards—Proximity cards—Part 3: Initialization and anticollision. ISO 144443-3:2011, International Organization for Standardization, 2011.

[19] JENKINS, M. A signal amplifier module for HF. `http://marcusjenkins.com/amateur-radio-2/a-signal-amplifier-module-for-hf/`. Acc.: 2015-06-19.

[20] KORTVEDT, H. Eavesdropping near field communication. In *NISK 2009*.

[21] LAURIE, A. RFIDler. `https://github.com/ApertureLabsLtd/RFIDler`. Acc.: 2015-06-19.

[22] MICROCHIP. *Antenna Circuit Design for RFID Applications*, 2003.

[23] MILLER, C. Exploring the NFC attack surface. Presented at Black Hat USA 2012.

[24] NOHL, K., EVANS, D., STARBUG, S., AND PLÖTZ, H. Reverse-engineering a cryptographic RFID tag. In *Proceedings of the 17th Conference on Security Symposium (USENIX Security 2008)*.

[25] NXP SEMICONDUCTORS. About MIFARE. `https://www.mifare.net/en/about-mifare/`. Acc.: 2015-06-19.

[26] NXP SEMICONDUCTORS. *Antenna design guide for MFRC52x, PN51x and PN53x*, 10 2010. Rev. 1.2.

[27] NXP SEMICONDUCTORS. *MIFARE Classic 1K - Mainstream contactless smart card IC for fast and easy solution development*, 2 2011. Rev. 3.1.

[28] NXP SEMICONDUCTORS. *NFC Forum Type 2 Tag compliant IC with 144 bytes user memory*, 2011. Rev. 3.2.

[29] NXP SEMICONDUCTORS. *MIFARE Ultralight contactless single-ticket IC*, 7 2014. Rev. 3.9.

[30] NXP SEMICONDUCTORS. *Standard 3V MIFARE reader solution*, 9 2014. Rev. 3.8.

[31] PENRI-WILLIAMS, K. E. Implementing an RFID 'Mifare Classic' attack. Master's thesis, City University London, 2009.

[32] PLÖTZ, H. RFID hacking. Presented at the 23rd Chaos Communication Congress (CCC 2006).

[33] PLÖTZ, H. Mifare classic – eine analyse der implementierung. Master's thesis, Humboldt-Universität, 2008.

[34] RICHTER, H., MOSTOWSKI, W., AND POLL, E. Fingerprinting passports. NLUUG spring conference on security (2008).

[35] RIGOL TECHNOLOGIES, INC. *MSO/DS2000A Series Digital Oscilloscope*.

[36] RYAN, R., ANDERSON, Z., AND CHIESA, A. Anatomy of a subway hack. Presented at the 16th DEFCON Hacking Conference (DEFCON 2008).

[37] TAN, W. H. Practical attacks on the MIFARE Classic. Master's thesis, Imperial College London, 2009.

[38] WEISS, M. Performing relay attacks on ISO 14443 contactless smart cards using NFC mobile equipment. Master's thesis, Technischen Universität München, 2010.

[39] WESTHUES, J. Proxmark 3. `https://github.com/Proxmark/proxmark3`. Acc.: 2015-06-19.

[40] ZHENG, Y., AND LI, M. ZOE: Fast cardinality estimation for large-scale RFID systems. In *INFOCOM* (2013).

Appendices

A MIFARE Cards

In this section, we discuss the memory layout and commands used by the MIFARE Ultralight (Section A.1) and Classic 1K cards (Section A.2), with more general ISO details in [18].

A.1 MIFARE Ultralight [29]

The MIFARE Ultralight has a 7-byte UID SN[0-6], including 2 check bytes BCC[0-1]. SN0 is the manufacturer ID (0x04 for NXP Semiconductors), and the check bytes are defined as follows: $BCC0 = 0x88 \oplus SN0 \oplus SN1 \oplus SN2$ and $BCC1 = SN3 \oplus SN4 \oplus SN5 \oplus SN6$. Lock bytes can be used to turn pages into read-only mode, while One-Time Pad (OTP) bytes can be set to 1, but never set back to 0 again. As can be seen in Figure 25, Ultralight cards go through 2 rounds of ANTICOLLISION and SELECT commands, and their memory layout is shown in Figure 26.

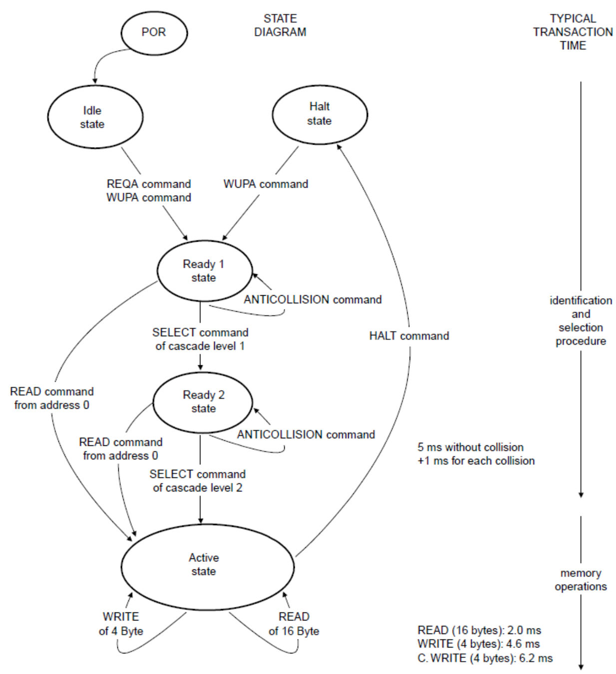

Figure 25. MIFARE Ultralight FSM

Page address		Byte number			
Decimal	Hex	0	1	2	3
0	00h	serial number			
1	01h	serial number			
2	02h	serial number	internal	lock bytes	lock bytes
3	03h	OTP	OTP	OTP	OTP
4 to 15	04h to 0Fh	user memory			

Figure 26. MIFARE Ultralight Memory Layout

Table 1 shows the possible commands when communicating with a MIFARE Ultralight card. The address for the READ and WRITE commands are between 0x00 and 0x0F and include roll-over C[0-1] is the CRC, and D[0-15] refers to data bytes.

Table 1. MIFARE Ultralight Commands

Command	Code	Response
REQA	0x26	0x44 0x00 (ATQA)
WUPA	0x52	0x44 0x00 (ATQA)
ANTICOLLISION (1)	0x93 0x[20-67]	0x88 SN0 SN1 SN2 BCC0
SELECT (1)	0x93 0x70 0x88 SN0 SN1 SN2 BCC0 C0 C1	0x04 C0 C1
ANTICOLLISION (2)	0x95 0x[20-67]	SN3 SN4 SN5 SN6 BCC1
SELECT (2)	0x95 0x70 SN3 SN4 SN5 SN6 BCC1 C0 C1	0x00 C0 C1
READ	0x30 [Addr] C0 C1	D0 D1 $\cdots$ D15 C0 C1
WRITE	0xA2 [Addr] D0 D1 D2 D3 C0 C1	[ACK/NAK]
COMPATIBILITY WRITE (1)	0xA0 [Addr] C0 C1	[ACK/NAK]
COMPATIBILITY WRITE (2)	D0 D1 $\cdots$ D15 C0 C1	[ACK/NAK]
HALT	0x50 0x00 C0 C1	[passive ACK/NAK]

A.2 MIFARE Classic 1K [27]

The memory layout for Classic 1K cards can be seen in Figure 27. The card does not have a globally-unique identifier, but instead uses a 4-byte Non-Unique Identifier (NUID) in block 0. Each sector has a block called the "trailer", which contains the two keys and the access bits. The layout for these access bits is shown in Figure 28, where CX_y is the X'th access bit for block y. These access bits are interpreted differently based on whether the block is a trailer block (Figure 29) or a data block (Figure 30).

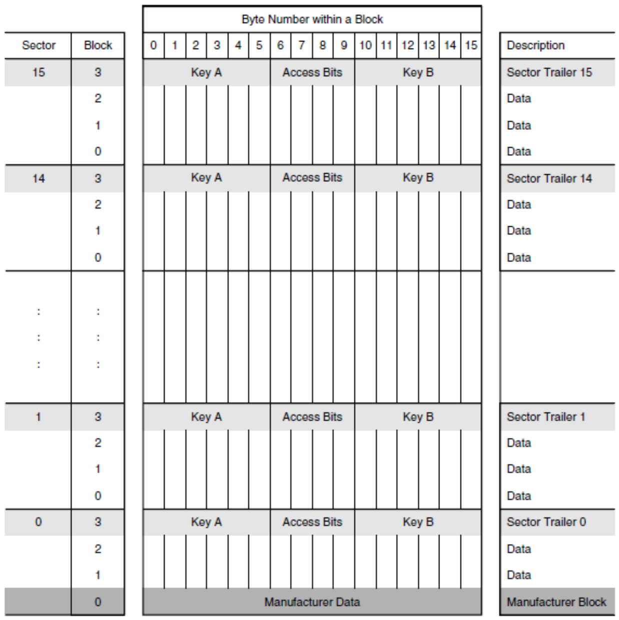

Figure 27. MIFARE Classic 1K Memory Layout

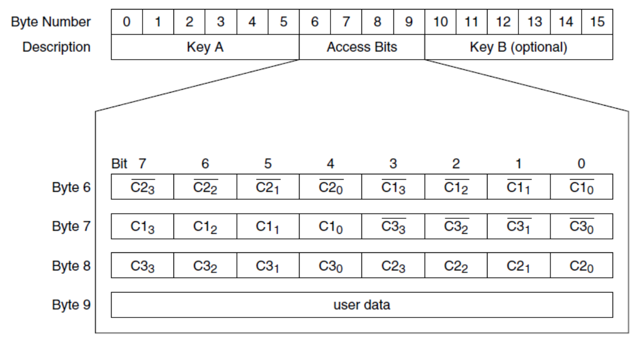

Figure 28. MIFARE Classic 1K Access Bits

Access bits			Access condition for						Remark
			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	Key B may be read[1]
0	1	0	never	never	key A	never	key A	never	Key B may be read[1]
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	Key B may be read, transport configuration[1]
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	

[1] for this access condition key B is readable and may be used for data

Figure 29. Trailer Block Access Conditions

Access bits			Access condition for				Application
C1	C2	C3	read	write	increment	decrement, transfer, restore	
0	0	0	key A|B[1]	key A|B1	key A|B1	key A|B1	transport configuration
0	1	0	key A|B[1]	never	never	never	read/write block
1	0	0	key A|B[1]	key B[1]	never	never	read/write block
1	1	0	key A|B[1]	key B[1]	key B[1]	key A|B[1]	value block
0	0	1	key A|B[1]	never	never	key A|B[1]	value block
0	1	1	key B[1]	key B[1]	never	never	read/write block
1	0	1	key B[1]	never	never	never	read/write block
1	1	1	never	never	never	never	read/write block

[1] if Key B may be read in the corresponding Sector Trailer it cannot serve for authentication (all grey marked lines in previous table). As a consequences, if the reader authenticates any block of a sector which uses the grey marked access conditions and using key B, the card will refuse any subsequent memory access after authentication.

Figure 30. Data Block Access Conditions

It is worth noting that the data blocks can be used for simple read/write operations, or they can be used as "value blocks" for applications which need more robustness and backups and could benefit from operations like INCREMENT and DECREMENT (see below). The layout for such blocks is shown in Figure 31.

Byte Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Description	value				$\overline{\text{value}}$				value				adr	$\overline{\text{adr}}$	adr	$\overline{\text{adr}}$

Figure 31. Value Block Layout

As indicated in Figure 32, the Classic 1K only uses a single round of ANTICOLLISION and SELECT commands, but includes a much more complicated 3-pass authentication mechanism. We explain in detail the encryption scheme in Appendix B, but we include the un-encrypted commands in Table 2, where NID[0-3] represents the NUID and $BCC = NID0 \oplus NID1 \oplus NID2 \oplus NID3$. Addresses range from 0x00 to 0x3F, while C[0-1] refers to the Checksum and D[0-15] to data bytes.

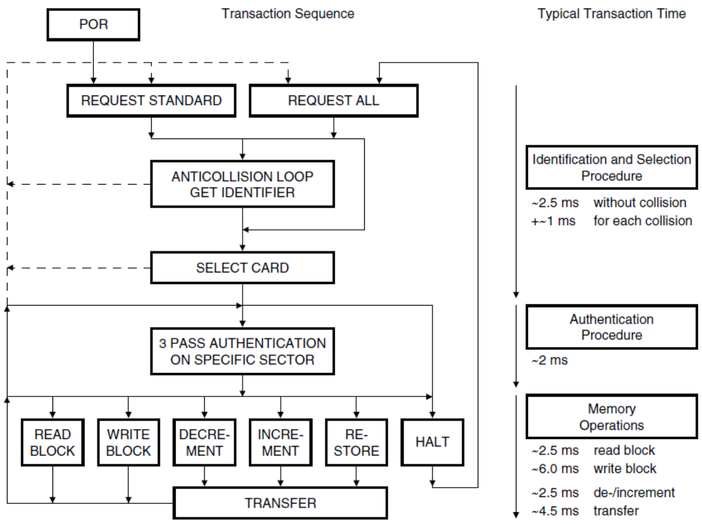

Figure 32. Classic 1K FSM

Table 2. MIFARE Classic 1K Plaintext Commands

Command	Code	Response
REQA	0x26	0x04 0x00 (ATQA)
WUPA	0x52	0x04 0x00 (ATQA)
ANTICOLLISION	0x93 0x20	NID0 NID1 NID2 NID3 BCC
SELECT	0x93 0x70 NID0 NID1 NID2 NID3 BCC C0 C1	0x08 C0 C1
AUTHA	0x60 [Addr] C0 C1	D0 D1 D2 D3 [TOKEN RB]
AUTHB	0x61 [Addr] C0 C1	D0 D1 D2 D3 [TOKEN RB]
AUTH3 [TOKEN AB]	D0 D1 ··· D7	D0 D1 D2 D3 [TOKEN BA]
READ	0x30 [Addr] C0 C1	D0 D1 ··· D15 C0 C1
WRITE (1)	0xA0 [Addr] C0 C1	[ACK/NAK]
WRITE (2)	D0 D1 ··· D15 C0 C1	[ACK/NAK]
INCREMENT (1)	0xC1 [Addr] C0 C1	[ACK/NAK]
DECREMENT (1)	0xC0 [Addr] C0 C1	[ACK/NAK]
RESTORE (1)	0xC2 [Addr] C0 C1	[ACK/NAK]
INC/DEC/RES (2)	D0 D1 ··· D15 C0 C1	[passive ACK/NAK]
TRANSFER	0xB0 [Addr] C0 C1	[ACK/NAK]
HALT	0x50 0x00 C0 C1	[passive ACK/NAK]

B CRYPTO1

In this section, we summarize the CRYPTO1 algorithm as reverse-engineered in [24, 12], but do not discuss the numerous vulnerabilities with the cipher which are addressed in the original papers. In the notation of [12], let the (unencrypted) nonce RB be denoted by n_T, the token AB be denoted as $\{n_R\},\{a_R\}$ and the token BA be denoted as $\{a_T\}$. Also denote by k the key, u the tag's (non) unique identifier (UID), and for any x, let x_i be its i-th bit (when this is well-defined). The CRYPTO1 algorithm uses a Linear Feedback Shift Register (LFSR) of size equal to 48 bits which is initialized by the key (also of length 48). Thus, denoting by $\alpha_i = a_i a_{i+1} \dots a_{i+47}$ the internal state of the LFSR at time i, we get that:

- $a_i := k_i$, for $0 \leq i \leq 47$
- $a_{48+i} := L(a_i, \dots, a_{47+i}) \oplus n_{T,i} \oplus u_i$, for $0 \leq i \leq 31$
- $a_{80+i} := L(a_{32+i}, \dots, a_{79+i}) \oplus n_{R,i}$, for $0 \leq i \leq 31$
- $a_{112+i} := L(a_{64+i}, \dots, a_{111+i}), \forall i \in \mathbb{N}$

where L is the LFSR "**feedback function**" defined by:

$$L(x_0 x_1 \dots x_{47}) := x_0 \oplus x_5 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{14} \oplus x_{15} \oplus x_{17} \oplus x_{19} \oplus x_{24} \oplus x_{25} \oplus x_{27} \oplus x_{29} \oplus x_{35} \oplus x_{39} \oplus x_{41} \oplus x_{42} \oplus x_{43}$$

The outputs are encrypted using a "**filter function**":

$f(x_0x_1\dots x_{47}) := f_c(f_a(x_9,x_{11},x_{13},x_{15}), f_b(x_{17},x_{19},x_{21},x_{23}), f_b(x_{25},x_{27},x_{29},x_{31}), f_a(x_{33},x_{35},x_{37},x_{39}), f_b(x_{41},x_{43},x_{45},x_{47}))$, where

$$f_a(y_0,y_1,y_2,y_3) := ((y_0 \vee y_1) \oplus (y_0 \wedge y_3)) \oplus (y_2 \wedge ((y_0 \oplus y_1) \vee y_3))$$
$$f_b(y_0,y_1,y_2,y_3) := ((y_0 \wedge y_1) \vee y_2) \oplus ((y_o \oplus y_1) \wedge (y_2 \vee y_3))$$
$$f_c(y_0,y_1,y_2,y_3,y_4) := (y_0 \vee ((y_1 \vee y_4) \wedge (y_3 \oplus y_4))) \oplus ((y_0 \oplus (y_1 \wedge y_3)) \wedge ((y_2 \oplus y_3) \vee (y_1 \wedge y_4)))$$

These two main functions are summarized in Figure 33.

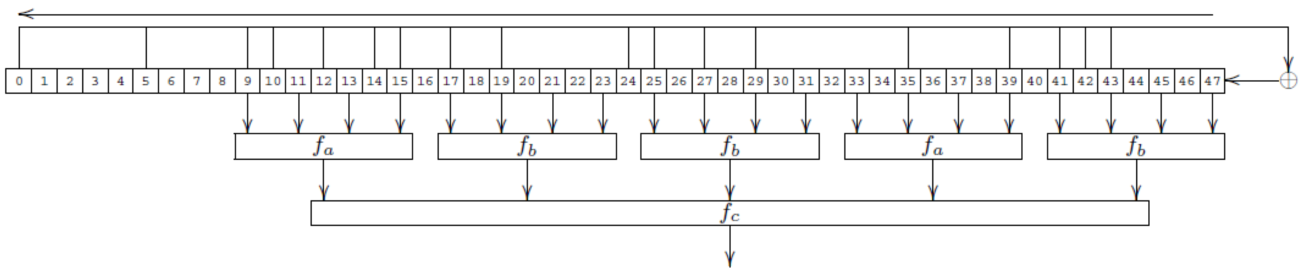

Figure 33. CRYPTO1 LFSR [12]

The keystream bit b_i is then defined by $b_i := f(a_i \dots a_{47+i})$, and the encryptions of the i-th regular bit (i.e. excluding start/end and parity bits) is defined by XORing with b_i. Specifically, for $0 \leq i \leq 31$, $\{n_{R,i}\} = n_{R,i} \oplus b_{32+i}$, $\{a_{R,i}\} = a_{R,i} \oplus b_{64+i}$, and $\{a_{T,i}\} = a_{T,i} \oplus b_{96+i}$. Note that the first 32 bits are not used in the first authentication, but they are used in all subsequent authentications, as the tag nonce is encrypted (and the cipher is re-initialized), so that $\{n_{T,i}\} = n_{T,i} \oplus b_i$.

The challenges only depend on the random nonce n_T, with $a_R = suc^{64}(n_T)$ and $a_T = suc^{96}(n_T)$, where the "**successor function**" — also used for the Random Number Generation (RNG) — is iteratively applied 64 and 96 times respectively. It is defined by $suc(x_0x_1\dots x_{31}) := x_1x_2\dots x_{31}(x_{16} \oplus x_{18} \oplus x_{19} \oplus x_{21})$ which only depends on the last 16 bits of the input.

It is worth noting that the start and ending transmission bits are not encrypted, but the parity bits are — by reusing the encryption bit for the next bit to be encrypted: $\{p_j\} := p_j \oplus b_{8j+8}$. This leaks information, and makes the protocol incompatible with the ISO standard, since the parity bits can be inverted. See Table 4 for an example.

C Example Traces

In this section we show two example traces, one for the MIFARE Ultralight (Table 3) and one for the Classic 1K (Table 4). For the latter, only part of the trace is shown. The key used is 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF, and the exclamation points indicate inverted parity bits.

Table 3. MIFARE Ultralight Trace

Direction	Bytes	Explanation
PCD→PICC	0x26	REQA
PICC→PCD	0x44 0x00	ATQA
PCD→PICC	0x93 0x20	ANTICOLLISION (1)
PICC→PCD	0x88 0x04 0xBE 0x6F 0x5D	ANTICOLLISION (1) RESPONSE 0x88 UID0 UID1 UID2 BCC0 $UID0 \oplus UID1 \oplus UID2 \oplus BCC0 = 0x88$
PCD→PICC	0x93 0x70 0x88 0x04 0xBE 0x6F 0x5D 0xA1 0x8E	SELECT (1) 0x88 + UID[0-2] + BCC0 CRC
PICC→PCD	0x04 0xDA 0x17	SELECT (1) RESPONSE CRC
PCD→PICC	0x95 0x20	ANTICOLLISION (2)
PICC→PCD	0x22 0x09 0x29 0x80 0x82	ANTICOLLISION (2) RESPONSE UID[3-6] + BCC1 $UID3 \oplus UID4 \oplus UID5 \oplus UID6 = BCC1$

PCD→PICC	0x95 0x70 0x22 0x09 0x29 0x80 0x82 0xD8 0xBA	SELECT(2) UID[3-6] + BCC1 CRC
PICC→PCD	0x00 0xFE 0x51	SELECT (2) RESPONSE CRC
PCD→PICC	0x30 0x00 0x02 0xA8	READ ADDR CRC
PICC→PCD	0x04 0xBE 0x6F 0x5D 0x22 0x09 0x29 0x80 0x82 0x48 0x00 0x00 0xE1 0x10 0x12 0x00 0xF8 0x99	UID UID UID + INTERNAL + LOCK BYTES OTP CRC
PCD→PICC	0x30 0x04 0x26 0xEE	READ ADDR CRC
PICC→PCD	0x01 0x03 0xA0 0x10 0x44 0x03 0x00 0xFE 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x81 0x3B	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x08 0x4A 0x24	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x0C 0x6E 0x62	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x50 0x00 0x57 0xCD	HALT CRC

Table 4. MIFARE Classic 1K Trace

Direction	Plaintext Bytes	Encrypted Bytes	Explanation
PCD→PICC	0x26	-	REQA
PICC→PCD	0x04 0x00	-	ATQA
PCD→PICC	0x93 0x20	-	ANTICOLLISION
PICC→PCD	0xCD 0x76 0x92 0x74 0x5D	-	ANTI RESP UID[0-3] BCC $\oplus UID = BCC$
PCD→PICC	0x93 0x70 0xCD 0x76 0x92 0x74 0x5D 0x45 0xDD	-	SELECT UID[0-3] + BCC CRC

PICC→PCD	0x08 0xB6 0xDD	-	SELECT RESPONSE CRC
PCD→PICC	0x60 0x3C 0x1A 0x80	-	AUTH (KEY A) ADDR CRC
PICC→PCD	0x0E 0x61 0x64 0xD6	-	n_T
PCD→PICC	0x15 0x45 0x90 0xA8 0x4F 0x4E 0x67 0x4E	0x78 0x5A 0x41 0x80! 0x50! 0x04! 0x8F 0x22!	n_R a_R
PICC→PCD	0x41 0x3E 0xEB 0xCF	0xCE! 0xCA! 0x0D! 0x83	a_T
PCD→PICC	0x30 0x3F 0x76 0x61	0x69! 0xAC! 0x4F! 0x02	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xD4 0x55	0xBC 0x2F 0xBD! 0xB1! 0x75! 0x44! 0x3C 0xD7! 0xD2 0x28 0x3B! 0xA5! 0x08 0x04 0x88! 0x18! 0x89 0x42!	KEY A (INACCESSIBLE) ACCESS BITS KEY B CRC
PCD→PICC	0x30 0x3E 0xFF 0x70	0x71! 0xF7 0x9F! 0x31	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0xE2! 0x99! 0x8D 0xE0 0x3F 0x96! 0xEF 0xC5 0xD0 0xD3! 0x24! 0x87! 0xF7 0x15! 0x06! 0x55! 0xA0! 0x97!	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x3D 0x64 0x42	0xC1! 0x43 0x22!0x92!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0xD2 0x00! 0x9D 0x87! 0xD9 0x0D 0x25 0x73 0x51 0x27 0x44 0xCC! 0x55 0x44! 0x85 0x9D 0x44 0xF6	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x3C 0xED 0x53	0x27 0x02 0x5C 0x41!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0x8B! 0xD1! 0xE3 0x87! 0x63! 0x75! 0x44 0x34 0x3B 0xAF! 0x27 0x0A! 0xAD! 0x84 0x1C 0xDB! 0xD8 0x4A	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x60 0x38 0x3E 0xC6	0xC6 0xDC 0xBA! 0x11!	(NESTED) AUTH (KEY A) ADDR CRC
PICC→PCD	0x8F 0x82 0x69 0x9E	0x70! 0xBD 0xED! 0x81	(ENCRYPTED) n_T
PCD→PICC	0x01 0x3A 0x6B 0xBA 0x73 0xD4 0x42 0x2D	0xFC! 0x1A 0x1A! 0x1D! 0x7D! 0x90 0x7E! 0x24!	n_R a_R
PICC→PCD	0xD0 0xA2 0x28 0xDB	0x87 0x4D 0xFF! 0x8A	a_T
PCD→PICC	0x30 0x3B 0x52 0x27	0x31 0x56 0xA1! 0x84	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xD4 0x55	0x64 0x85! 0x16 0x6D 0xCF! 0xF7 0x3C! 0x62 0xD2 0xB4 0x3B! 0x5F! 0xA8! 0x71 0xD1 0x6B! 0x6C! 0x42	KEY A (INACCESSIBLE) ACCESS BITS KEY B CRC

PCD→PICC	0x30 0x3A 0xDB 0x36	0x02! 0x44! 0xD4! 0x47!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0x41 0xBC! 0x42! 0xD9 0x39 0x4C! 0x9A 0x80! 0x4E 0x76 0xB1! 0xA1! 0xA4 0xD1 0x82! 0x61 0x28 0xBC!	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x9 0x40 0x04	0xA5 0x63 0x90! 0x32!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0xBD! 0xC3 0xA6! 0x15 0x8B 0x2A! 0x6A! 0x03 0x72! 0xEF! 0x02! 0x38 0x05! 0xC5! 0x3F 0x4E! 0x94! 0x08	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x30 0x38 0xC9 0x15	0xE2 0x1C! 0x8A 0x16!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0x51 0x1B 0xA6! 0x49 0x29! 0xF2 0x75 0x35! 0x1B! 0xE1 0x72 0x68 0x7F 0x3F 0x2A 0xE9! 0x73 0x0F	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC
PCD→PICC	0x60 0x34 0x52 0x0C	0x7F! 0xB8! 0x88 0x7B!	(NESTED) AUTH (KEY A) ADDR CRC
PICC→PCD	0xDC 0xFC 0x96 0x2B	0x23! 0x23! 0x6E 0xF4!	(ENCRYPTED) n_T
PCD→PICC	0xEE 0x08 0xB0 0x0A 0xF6 0x01 0xBA 0x11	0x1A 0xB9! 0xEF! 0x7B! 0xC5 0xC3 0x51 0x57!	n_R a_R
PICC→PCD	0x6E 0x27 0x63 0x93	0x3E 0x19 0x48! 0xF4!	a_T
PCD→PICC	0x30 0x37 0x3E 0xED	0xD6! 0x59! 0xB4! 0x73!	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xD4 0x55	0x32! 0x84 0xE6 0xB1! 0x45! 0x56! 0x5C! 0x1A! 0xED! 0xD3! 0xED! 0x76 0x32! 0x5F 0x5D! 0x4D 0x96! 0xFC!	KEY A (INACCESSIBLE) ACCESS BITS KEY B CRC
PCD→PICC	0x30 0x36 0xB7 0xFC	0xBB! 0x4D! 0xF8 0x55	READ ADDR CRC
PICC→PCD	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x37 0x49	0x7E 0x9E! 0x34! 0x38! 0xDC! 0xE2! 0xF9 0x98! 0xA0 0x88 0x78! 0xA9! 0xCD! 0xEE! 0x17 0xD8 0xFF! 0xFD!	USER DATA (1) USER DATA (2) USER DATA (3) USER DATA (4) CRC