The Mifare Classic Authentication Process

Nick Holdren

Tag Selection

On start up, the reader sends out command Req Type A (26). This procedure searches the field of the reader for tags of Type A. Type A cards have specific anti-collision procedure that will be detailed later. Tags within the field respond with Answer Req (04 00) to the reader letting it know that is of Type A. At this point, the reader cannot assume that there is only one tag in the field. Therefore, the reader sends out a select command (93 20) (20 tells reader to strictly read uids that are transmitted).

Tags within the field send their complete uid to the reader. We assume that tags within the field have their own unique serial numbers. If more than one tag responds then a collision will occur. Type A tags handle anti-collision by doing a bit by bit comparison of the uids that respond. The uids that respond are XOR-ed to determine the number of valid bits before the collision. For example:

uid_A : 1111
uid_B : 1101

The result would be that there are two valid bits before a collision occurs. The position of the collision is noted by the number of valid bits (NVB). Then either a 1 or 0 is appended to the segment of the valid bits. The choice between 0 and 1 is decided by the reader. The reader then sends out a select command with requirement of the NVB along with the valid bits. Then tags that meet the requirement of matching the first NVB respond. Again, if there is a collision then the process repeats itself up to 32 times until a single tag is selected.

Once there are no more collisions the reader sends out the Select command again (93 70). In this case NVB is 70, which tells the reader to send the entire uid. Therefore, the select command is given followed by the entire uid of the valid uid which was determined by the anti-collision procedure. The uid is then followed by a checksum or checkbyte, which is computed by XOR-ing the last 4 bytes of the uid. Checksums are used to verify that the data was successfully transmitted. If the checksum does not match the computed checksum using the transmitted uid then it can be determined that there was an error during transmission.

If a tag is matched it sends a select acknowledge response informing the reader that it has been selected. The selected tag then moves from a ready state to an active state and the procedure continues to the authentication process.

Authentication

Generating $n_T$
In order to authenticate the reader sends a request to the tag Auth (block 30) (60 30). 60 represents the call for authentication and 30 the block that is to be accessed. The tag sends a challenge to the reader, $n_T$, which is generated by a 16 bit Linear Feedback Shift Register (LFSR) with a pseudo-random generator defined by the polynomial $x^{16} + x^{14} + x^{13} + x^{11} + 1$. The exponents denote the taps or positions of the bits that are to be fed back into the register. The challenge $n_T$ has a length of 32 bits, but as mentioned before the LFSR is only 16 bits. This means that the 16 bits that are not accounted for are generated by the LFSR from the initial seed and . The value of the LFSR updates with every clock tick (every 9.44 $\mu$s) and value depends on the time the tag powers up and when communication is invoked. Therefore, the challenge nonce, $n_T$, since LFSR of the tag is deterministic, can be replicated at anytime by controlling start up and when communications begins. Initially, the state of the LFSR is only the input bit 1 and the rest of the register are 0 s. With each clock tick, the feedback bits are computed using the specified taps and XOR-ing them to generate a new input bit. This bit is fed back into the register onto the right and the register is shifted to the left. The operation can be defined as:

$$x^n (x^{16} + x^{14} + x^{13} + x^{11} + 1)$$

or

$$L_{16}(x_0, x_1, ..., x_{15}) = x_0 \oplus x_1 \oplus x_3 \oplus x_{13} \oplus x_{14} \oplus x_{16}$$

The psuedo random generator operates in a finite field $\mathbb{F}_2$. Where $\mathbb{F}_2$ is the field of 2 elements $\{0, 1\}$. $\oplus$ denotes the XOR operation, which is a logical operation in general described a $p \neq q$. Using the XOR operation in $\mathbb{F}_2$ generates a value in $\mathbb{F}_2$. The truth table of $p \oplus q$ is as follows:

- $1 \oplus 1 = 0$
- $1 \oplus 0 = 1$
- $0 \oplus 1 = 1$
- $0 \oplus 0 = 0$

**Validating Authentication Through the Successor of $n_T$**

Once a proper nonce has been generated, one consisting of 32 bits, it is sent as challenge $n_T$ to the reader unencrypted (in plaintext.) Both the reader and the tag now have $n_T$ and uid, and they both have the necessary seed to begin to initialize the CRYPTO-1 cipher as later described. At this point we can assume that both the tag and the reader have the same cipher state. In order to answer/check the challenges it is only necessary for the tag and reader to be able to generate the successors of $n_T$ denoted suc($n_T$) and described by:

$$suc(x_0, x_1, ..., x_{31}) = x_1 x_2 ... x_{31} L_{16}(x_0, x_1, ..., x_{15})$$

As found in section 4 of [GKM+08] $a_R = suc^2(n_T)$ or second successor and $a_T = suc^3(n_T)$ or third successor, which means that if after decryption $a_R$ equals what the tag has generated as suc$^2(n_T)$ and if after decryption $a_T$ equals what the reader has generated as suc$^3(n_T)$ then authentication is complete.

**Summary of Authentication Process**

To begin authentication tag sends $n_T$ unencrypted to the reader. Both devices having $n_T$, uid, and shared key (K) are able to begin to initialize the cipher. The reader generates $n_R$ using it's psuedo random generator (dependent on invocations) and sends it with ks1 to the tag afterwhich it feeds in $n_R$ into the cipher. Once $n_R$ is fed in the initialization of the reader's cipher is complete. Having all resources at hand (uid, $n_T$, $n_R \oplus$ ks1) the tag, in the same manner, is able to initialize its cipher by first
inputting $n_T \oplus \text{uid}$. And due to the fact that $\text{ks}1$ is generated at the point in which $n_R$ is fed into the cipher (immediately following the entry of the 32 bit $n_T \oplus \text{uid}$), when the tag inputs $n_R \oplus \text{ks}1$, $n_R$ comes out because $\text{ks}1 \oplus (n_R \oplus \text{ks}1) = n_R$. The tag then shifts in $n_R$ and initialization is now complete for the tag.

During transmission, $n_R \oplus \text{ks}1$ is sent along with $a_R \oplus \text{ks}2$ and again the tag runs $a_R \oplus \text{ks}2$ through it's cipher to decrypt/compute $\text{ks}2 \oplus (a_R \oplus \text{ks}2) = a_R$. The tag can then verify that $a_R = \text{suc}^2(n_T)$. The tag sends $a_T \oplus \text{ks}3$ to the reader and the reader receives $a_T \oplus \text{ks}3$, which it runs through it's cipher and is able to verify through the fact $\text{ks}3 \oplus (a_T \oplus \text{ks}3) = a_T$ that indeed $a_T = \text{suc}^3(n_T)$. If the correct values are calculated then authentication continues at each step, otherwise an error message is sent.

**Initialization of CRYPTO - 1 and Generating $n_R$**

As mentioned before the pseudorandom generator in the reader updates on every invocation not on clock ticks, but otherwise it functions in same manner in order to generate $n_R$. Knowing that the LFSR updates at every invocation allows us to control the value of $n_R$ by turning off the reader, starting it up again and simply keeping track of the number of invocations thus we can control or track the value of $n_R$ to an extent. Unlike $n_T$ however, $n_R$ is encrypted with $\text{ks}1$ using the CRYPTO-1 cipher before it is transmitted and therefore is not easily obtained. The CRYPTO-1 cipher is a LFSR with generating polynomial:

$$g(x) = x^{48} + x^{43} + x^{39} + x^{38} + x^{36} + x^{34} + x^{33} + x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^{9} + x^{7} + x^{6} + x^{5} + 1$$

or

$$G(x_0, x_1, x_3, ... x_{47}) = x_0 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{12} \oplus x_{19} \oplus x_{21} \oplus x_{23} \oplus x_{24} \oplus x_{29} \oplus x_{31} \oplus x_{33} \oplus x_{34} \oplus x_{36} \oplus x_{38} \oplus x_{39} \oplus x_{43}$$

Every clock tick the register is shifted to the left and the leftmost bit is removed and fed into the register on the right. When the CRYPTO - 1 cipher is initialized the shared key (K) is the initial state of the cipher. Next, $n_T$ XOR uid is shifted into the register bitwise such that $(n_T, i \text{ XOR \ uid})$ XOR $k_i$. Then, $n_R$ is fed into register bitwise $n_R$ XOR cipher. The entire initialization procedure can be described by:

**Initial State:**

$$C_i = K_i \forall \ i \in [0, 47]$$

The initial state $C_i$ of the cipher is exactly the shared key (K) by itself. Next, $n_T \oplus \text{uid}$ is fed into the cipher where it XOR-ed with feedback function $G(x_0, x_1, x_3, ... x_{47})$:

$$C_{48+i} = G(x_0, x_1, x_3, ... x_{47}) \oplus (n_T \oplus \text{uid}) \quad \forall \ i \in [0, 31]$$

Once complete feed in $n_R$ and initialization is complete,

$$C_{80+i} = G(x_{32+i}, ..., x_{79+i}) \oplus n_R, i \quad \forall \ i \in [0, 31]$$

The initialized LFSR operates as normal by using the feedback function:

$$C_{112+i} = G(x_{64+i}, ..., x_{111+i}) \quad \forall \ i \in N$$

This procedure is done by both the tag and the reader, although instead of the tag generating $n_R$ it must first decrypt it and then feed it into the cipher and thus they both reach the same cipher state.

**Generating the Keystream**

In order to encrypt a keystream must be generated to be XOR-ed with each nonce. The following describes the procedure:
Given the key stream bit \( b_i \in \mathbb{F}_2 \) at time \( i \) \( b_i = f(a_i, a_{i+1}, \ldots, a_{i+16}) \) where \( f \) is the filter function:

\[
f(x_{01} \ldots x_{45}):= f_\alpha(x_{09}, x_{11}, x_{13}, x_{15}), f_\beta(x_{17}, x_{19}, x_{21}, x_{23}), f_\gamma(x_{25}, x_{27}, x_{29}, x_{31}), f_\delta(x_{33}, x_{35}, x_{37}, x_{39}), f_\epsilon(x_{41}, x_{43}, x_{45}, x_{47})
\]

where \( x_i \) represents the bit position in the cipher and \( f_\alpha, f_\beta, f_\gamma \) are the filter layers in CRYPTO-1 and are defined by:

\[
f_\alpha(y_0, y_1, y_2, y_3) = ((y_0 \lor y_1) \oplus (y_0 \land y_3)) \oplus (y_2 \land ((y_0 \lor y_1) \lor y_3))
\]

\[
f_\beta(y_0, y_1, y_2, y_3) = ((y_0 \land y_1) \lor y_2) \oplus ((y_0 \land y_1) \land (y_2 \land y_3))
\]

\[
f_\gamma(y_0, y_1, y_2, y_3, y_4) = (y_0 \lor ((y_1 \lor y_4) \land (y_3 \land y_4))) \oplus ((y_2 \lor (y_1 \land y_3)) \land (y_2 \land y_3) \lor (y_1 \land y_4))
\]

with logical operators \( \land \) (and), \( \lor \) (or), and \( \oplus \) The bits from the cipher are inputted into the various filters that then returns a single keystream bit. This bit is then XOR-ed bitwise with the information that is going to be sent. In this case, it is for sending \( n_R \) and so we have:

\[
\{n_{R,i}\} = n_{R,i} \oplus b_{32+i}, \forall i \in [0,31] \quad \text{At the 32}^{\text{nd}} \text{bit of the key stream } n_R \text{ begins it's encryption with } b_{32+i}, \text{known as ks1.}
\]

\[
\{a_{R,i}\} = a_{R,i} \oplus b_{64+i}, \forall i \in [0,31] \quad \text{At the 64}^{\text{th}} \text{bit of the key stream } a_R \text{ begins it's encryption with } b_{64+i}, \text{known as ks2.}
\]

And likewise the tag encrypts \( a_T \):

\[
\{a_{T,i}\} = a_{T,i} \oplus b_{96+i}, \forall i \in [0,31] \quad \text{At the 96}^{\text{th}} \text{bit of the key stream } a_T \text{ begins it's encryption with } b_{96+i}, \text{known as ks3.}
\]

And due to the fact that both the tag and reader are set to the same internal cipher state they are capable of encrypting and decrypting any information sent between each other.

References


