Dismantling MIFARE Classic

Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers,
Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{flaviog,petervr,ronny,bart}@cs.ru.nl
{gkoningg,rmuijrer,rverdult}@sci.ru.nl

Abstract. The MIFARE Classic is a contactless smart card that is used extensively in access control for office buildings, payment systems for public transport, and other applications. We reverse engineered the security mechanisms of this chip: the authentication protocol, the symmetric cipher, and the initialization mechanism. We describe several security vulnerabilities in these mechanisms and exploit these vulnerabilities with two attacks; both are capable of retrieving the secret key from a genuine reader. The most serious one recovers the secret key from just one or two authentication attempts with a genuine reader in less than a second on ordinary hardware and without any pre-computation. Using the same methods, an attacker can also eavesdrop the communication between a tag and a reader, and decrypt the whole trace, even if it involves multiple authentications. This enables an attacker to clone a card or to restore a real card to a previous state.

1 Introduction

Over the last few years, more and more systems adopted RFID and contactless smart cards as replacement for bar codes, magnetic stripe cards and paper tickets for a wide variety of applications. Contactless smart cards consist of a small piece of memory that can be accessed wirelessly, but unlike RFID tags, they also have some computing capabilities. Most of these cards implement some sort of simple symmetric-key cryptography, making them suitable for applications that require access control to the smart card's memory.

A number of large-scale applications make use of contactless smart cards. For example, they are used for payment in several public transport systems like the Oyster card[1] in London and the OV-Chipkaart[2] in The Netherlands, among others. Many countries have already incorporated a contactless smart card in their electronic passports [HHJ+06]. Many office buildings and even secured facilities like airports and military bases use contactless smart cards for access control.

There is a huge variety of cards on the market. They differ in size, casing, memory, and computing power. They also differ in the security features they provide.

[1] http://oyster.tfl.gov.uk
[2] http://www.ov-chipkaart.nl

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 97–114, 2008.

A well known and widely used system is MIFARE. This is a product family from NXP Semiconductors (formerly Philips Semiconductors), currently consisting of four different types of cards: Ultralight, Classic, DESFire and SmartMX. According to NXP, more than 1 billion MIFARE cards have been sold and there are about 200 million MIFARE Classic tags in use around the world, covering about 85% of the contactless smart card market. Throughout this paper we focus on this tag. MIFARE Classic tags provide mutual authentication and data secrecy by means of the so called CRYPTO1 cipher. This is a stream cipher using a 48 bit secret key. It is proprietary of NXP and its design is kept secret.

Our Contribution. This paper describes the reverse engineering of the MIFARE Classic chip. We do so by recording and studying traces from communication between tags and readers. We recover the encryption algorithm and the authentication protocol. It also unveils several vulnerabilities in the design and implementation of the MIFARE Classic chip. This results in two attacks that recover a secret key from a MIFARE reader.

The first attack uses a vulnerability in the way the cipher is initialized to split the 48 bit search space in a k bit online search space and $48-k$ bit offline search space. To mount this attack, the attacker needs to gather a modest amount of data from a genuine reader. Once this data has been gathered, recovering the secret key is as efficient as a lookup operation on a table. Therefore, it is much more efficient than an exhaustive search over the whole 48 bit key space.

The second and more efficient attack uses a cryptographic weakness of the CRYPTO1 cipher allowing us to recover the internal state of the cipher given a small part of the keystream. To mount this attack, one only needs one or two partial authentication from a reader to recover the secret key within one second, on ordinary hardware. This attack does not require any pre-computation and only needs about 8 MB of memory to be executed.

When an attacker eavesdrops communication between a tag and a reader, the same methods enable us to recover all keys used in the trace and decrypt it. This gives us sufficient information to read a card, clone a card, or restore a card to a previous state. We have successfully executed these attacks against real systems, including the London Oyster Card and the Dutch OV-Chipkaart.

Related Work. De Koning Gans, Hoepman and Garcia [KHG08] proposed an attack that exploits the malleability of the CRYPTO1 cipher to read partial information from a MIFARE Classic tag. Our paper differs from [KHG08] since the attacks proposed here focus on the reader.

Nohl and Plötz have partly reverse engineered the MIFARE Classic tag earlier [NP07], although not all details of their findings have been made public. Their research takes a very different, hardware oriented, approach. They recovered the algorithm, partially, by slicing the chip and taking pictures with a microscope. They then analyzed these pictures, looking for specific gates and connections.

Their presentation has been of great stimulus in our discovery process. Our approach, however, is radically different as our reverse engineering is based on the study of the communication behavior of tags and readers. Furthermore,

the recovery of the authentication protocol, the cryptanalysis, and the attacks presented here are totally novel.

Overview. In Section 2 we briefly describe the hardware used to analyze the MIFARE Classic. Section 3 summarizes the logical structure of the MIFARE Classic. Section 4 then describes the way a tag and a reader authenticate each other. It also details how we reverse engineered this authentication protocol and points out a weakness in this protocol enabling an attacker to discover 64 bits of the keystream. Section 5 describes how we recovered the CRYPTO1 cipher by interacting with genuine readers and tags. Section 6 then describes four concrete weaknesses in the authentication protocol and the cipher and how they can be exploited. Section 7 describes how this leads to concrete attacks against a reader. Section 8 shows that these attacks are also applicable if the reader authenticates for more than a single block of memory. Section 9 describes consequences and conclusions.

2 Hardware Setup

For this experiment we designed and built a custom device for tag emulation and eavesdropping. This device, called Ghost, is able to communicate with a contactless smart card reader, emulating a tag, and eavesdrop communication between a genuine tag and reader. The Ghost is completely programmable and is able to send arbitrary messages. We can also set the `uid` of the Ghost which is not possible with manufacturer tags. The hardware cost of the Ghost is approximately €40. We also used a ProxMark[3], a generic device for communication with RFID tags and readers, and programmed it to handle the ISO14443-A standard. As it provides similar functionality to the Ghost, we do not make a distinction between these devices in the remainder of the paper.

On the reader side we used an OpenPCD reader[4] and an Omnikey reader[5]. These readers contain a MIFARE chip implementing the CRYPTO1 cipher and are fully programmable.

Notation. In MIFARE, there is a difference between the way bytes are represented in most tools and the way they are being sent over the air. The former, consistent with the ISO14443 standard, writes the most significant bit of the byte on the left, while the latter writes the least significant bit on the left. This means that most tools represent the value `0x0a0b0c` as `0x50d030` while it is sent as `0x0a0b0c` on the air. Throughout this paper we adopt the latter convention (with the most significant bit left, since that has nicer mathematical properties) everywhere except when we show traces so that the command codes are consistent with the ISO standard.

Finally, we number bits (in keys, nonces, and cipher states) from left to right, starting with 0. For data that is transmitted, this means that lower numbered bits are transmitted before higher numbered bits.

[3] `http://cq.cx/proxmark3.pl,http://www.proxmark.org`
[4] `http://www.openpcd.org`
[5] `http://omnikey.aaitg.com`

3 Logical Structure of the MIFARE Classic Tags

The MIFARE Classic tag is essentially an EEPROM memory chip with secure communication provisions. Basic operations like read, write, increment and decrement can be performed on this memory. The memory of the tag is divided into sectors. Each sector is further divided into blocks of 16 bytes each. The last block of each sector is called the sector trailer and stores two secret keys and access conditions corresponding to that sector.

To perform an operation on a specific block, the reader must first authenticate for the sector containing that block. The access conditions of that sector determine whether key A or B must be used. Figure 1 shows a schematic of the logical structure of the memory of a MIFARE Classic tag.

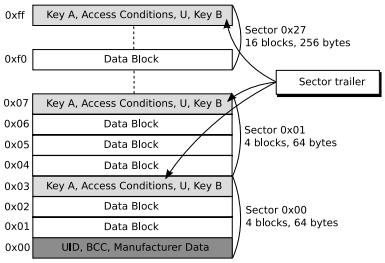

Fig. 1. Logical structure

4 Authentication Protocol

When the tag enters the electromagnetic field of the reader and powers up, it immediately starts the anti-collision protocol by sending its uid. The reader then selects this tag as specified in ISO14443-A [ISO01].

According to the manufacturer's documentation, the reader then sends an authentication request for a specific block. Next, the tag picks a challenge nonce n_T and sends it to the reader in the clear. Then the reader sends its own challenge nonce n_R together with the answer a_R to the challenge of the tag. The tag finishes authentication by replying a_T to the challenge of the reader. Starting with n_R, all communication is encrypted. This means that n_R, a_R, and a_T are XOR-ed with the keystream $\mathsf{ks}_1, \mathsf{ks}_2, \mathsf{ks}_3$. Figure 2 shows an example.

Step	Sender	Hex	Abstract
01	Reader	26	req type A
02	Tag	04 00	answer req
03	Reader	93 20	select
04	Tag	c2 a8 2d f4 b3	uid,bcc
05	Reader	93 70 c2 a8 2d f4 b3 ba a3	select(uid)
06	Tag	08 b6 dd	MIFARE 1k
07	Reader	60 30 76 4a	auth(block 30)
08	Tag	42 97 c0 a4	n_T
09	Reader	7d db 9b 83 67 eb 5d 83	$n_R \oplus \mathsf{ks}_1, a_R \oplus \mathsf{ks}_2$
10	Tag	8b d4 10 08	$a_T \oplus \mathsf{ks}_3$

Fig. 2. Authentication Trace

We started experimenting with the Ghost and an OpenPCD reader which we control. The pseudo-random generator in the tag is fully deterministic. Therefore the nonce it generates only depends on the time between power up and the start of communication [NP07]. Since we control the reader, we control this timing and therefore can get the same tag nonce every time. With the Ghost operating as a tag, we can choose custom challenge nonces and uids. Furthermore, by fixing n_T (and uid) and repeatedly authenticating, we found out that the reader produces the same sequence of nonces every time it is restarted. Unlike in the tag, the state of the pseudo-random generator in the reader does not update every clock tick but with every invocation.

The pseudo-random generator in the tag used to generate n_T is a 16 bit LFSR with generating polynomial $x^{16}+x^{14}+x^{13}+x^{11}+1$. Since nonces are 32 bits long and the LFSR has a 16 bit state, the first half of n_T determines the second half. This means that given a 32 bit value, we can tell if it is a proper tag nonce, i.e., if it could be generated by this LFSR. To be precise, a 32 bit value $n_0 n_1 \dots n_{31}$ is a proper tag nonce if and only if $n_k \oplus n_{k+2} \oplus n_{k+3} \oplus n_{k+5} \oplus n_{k+16} = 0$ for all $k \in \{0, 1, \dots, 15\}$. Remark that the Ghost can send arbitrary values as nonces and is not restricted to sending proper tag nonces.

Experimenting with authentication sessions with various uids and tag nonces, we noticed that if $n_T \oplus \mathsf{uid}$ remains constant, then the ciphertext of the encrypted reader nonce also remains constant. The answers a_T and a_R, however, have different ciphertexts in the two sessions. For example, in Figure 2 the uid is `0xc2a82df4` and n_T is `0x4297c0a4`, therefore $n_T \oplus \mathsf{uid}$ is `0x803fed50`. If we instead take uid to be `0x1dfbe033` and n_T to be `0x9dc40d63`, then $n_t \oplus \mathsf{uid}$ still equals `0x803fed50`. In both cases, the encrypted reader nonce $n_R \oplus \mathsf{ks}_1$ is `0x7ddb9b83`. However, in Figure 2, $a_R \oplus \mathsf{ks}_2$ is `0x67eb5d83` and $a_T \oplus \mathsf{ks}_3$ is `0x8bd41008`, while with the modified uid and n_T they are, respectively, `0x4295c446` and `0xeb3ef7da`.

This suggests that the keystream in both runs is the same and it also suggests that a_T and a_R depend on n_T. By XOR-ing both answers $a_R \oplus \mathsf{ks}_2$ and $a'_R \oplus \mathsf{ks}_2$ together we get $a_R \oplus a'_R$. We noticed that $a_R \oplus a'_R$ is a proper tag nonce. Because the set of proper tag nonces is a linear subspace of $\mathbb{F}_2^{32}$, where $\mathbb{F}_2$ is the field of two elements, the XOR of proper tag nonces is also a proper tag nonce. This suggests that a_R and a'_R are also proper tag nonces.

Given a 32 bit nonce n_T generated by the LFSR, one can compute the successor $\mathsf{suc}(n_T)$ consisting of the next 32 generated bits. At this stage we could verify that $a_R \oplus a'_R = \mathsf{suc}^2(n_T \oplus n'_T) = \mathsf{suc}^2(n_T) \oplus \mathsf{suc}^2(n'_T)$ which suggests that $a_R = \mathsf{suc}^2(n_T)$ and $a'_R = \mathsf{suc}^2(n'_T)$. Similarly for the answer from the tag we could verify that $a_T = \mathsf{suc}^3(n_T)$ and $a'_T = \mathsf{suc}^3(n'_T)$.

Summarizing, the authentication protocol can be described as follows; see Figure 3. After the nonce n_T is sent by the tag, both tag and reader initialize the cipher with the shared key K, the uid, and the nonce n_T. The reader then picks its challenge nonce n_R and sends it encrypted with the first part of the keystream ks_1. Then it updates the cipher state with n_R. The reader authenticates by sending $\mathsf{suc}^2(n_T)$ encrypted, i.e., $\mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$. At this point the tag is able

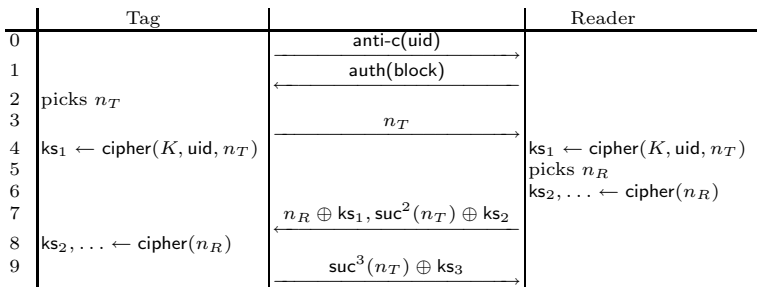

Fig. 3. Authentication Protocol

to update the cipher state in the same way and verify the authenticity of the reader. The remainder of the keystream $\mathsf{ks}_3, \mathsf{ks}_4 \ldots$ is now determined and from now on all communication is encrypted, i.e., XOR-ed with the keystream. The tag finishes the authentication protocol by sending $\mathsf{suc}^3(n_T) \oplus \mathsf{ks}_3$. Now the reader is able to verify the authenticity of the tag.

4.1 Known Plaintext

From the description of the authentication protocol it is easy to see that parts of the keystream can be recovered. Having seen n_T and $\mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$, one can recover ks_2 (i.e., 32 bits of keystream) by computing $\mathsf{suc}^2(n_T)$ and XOR-ing.

Moreover, experiments show that if in step 9 of the authentication protocol the tag does not send anything, then most readers will time out and send a halt command. Since communication is encrypted it actually sends $\mathsf{halt} \oplus \mathsf{ks}_3$. Knowing the byte code of the halt command (`0x500057cd` [ISO01]) we recover ks_3.

Some readers do not send a halt command but instead continue as if authentication succeeded. This typically means that it sends an encrypted read command. As the byte code of the read command is also known [KHG08], this also enables us to recover ks_3 by guessing the block number.

It is important to note that one can obtain such an authentication session (or rather, a partial authentication session, as the Ghost never authenticates itself) from a reader (and hence ks_2, ks_3) without knowing the secret key and, in fact, without using a tag.

If an attacker does have access to both a tag and a reader and can eavesdrop a successful (complete) authentication session, then both ks_2 and ks_3 can be recovered from the answers $\mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$ and $\mathsf{suc}^3(n_T) \oplus \mathsf{ks}_3$ of the tag and the reader. This works even if the reader does not send halt or read after timeout.

5 CRYPTO1 Cipher

The core of the CRYPTO1 cipher is a 48-bit linear feedback shift register (LFSR) with generating polynomial $g(x) = x^{48} + x^{43} + x^{39} + x^{38} + x^{36} + x^{34} + x^{33} +$

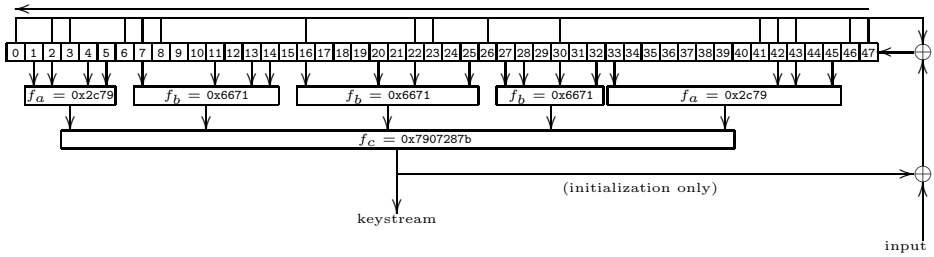

Fig. 4. The Hitag2 Cipher

$x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^9 + x^7 + x^6 + x^5 + 1$. This polynomial was given in [NESP08]; note it can also be deduced from the relation between uid and the secret key described in [NP07]. At every clock tick the register is shifted one bit to the left. The leftmost bit is discarded and the feedback bit is computed according to $g(x)$. Additionally, the LFSR has an input bit that is XOR-ed with the feedback bit and then fed into the LFSR on the right. To be precise, if the state of the LFSR at time k is $r_k r_{k+1} \dots r_{k+47}$ and the input bit is i, then its state at time $k+1$ is $r_{k+1} r_{k+2} \dots r_{k+48}$, where

$$r_{k+48} = r_k \oplus r_{k+5} \oplus r_{k+9} \oplus r_{k+10} \oplus r_{k+12} \oplus r_{k+14} \oplus r_{k+15} \oplus r_{k+17} \oplus r_{k+19} \oplus r_{k+24} \oplus r_{k+27} \oplus r_{k+29} \oplus r_{k+35} \oplus r_{k+39} \oplus r_{k+41} \oplus r_{k+42} \oplus r_{k+43} \oplus i. \quad (1)$$

The input bit i is only used during initialization.

To encrypt, selected bits of the LFSR are put through a filter function f. Exactly which bits of the LFSR are put through f and what f actually is, was not revealed in [NP07]. Note that the general structure of CRYPTO1 is very similar to that of the Hitag2. This is a low frequency tag from NXP; the description of the cipher used in the Hitag2 is available on the Internet[6]. We used this to make educated guesses about the details of the initialization of the cipher (see Section 5.1 below) and about the details of the filter function f (see Section 5.2 below).

5.1 Initialization

The LFSR is initialized during the authentication protocol. As before, we experimented running several authentication sessions with varying parameters. As we mention in Section 4, if $n_T \oplus$ uid remains constant, then the encrypted reader nonce also remains constant. This suggests that $n_T \oplus$ uid is first fed into the LFSR. Moreover, experiments showed that, if special care is taken with the

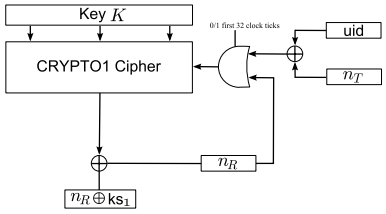

Fig. 5. Initialization Diagram

[6] http://cryptolib.com/ciphers/hitag2/

feedback bits, it is possible to modify $n_T \oplus \mathsf{uid}$ and the secret key K in such a way that the ciphertext after authentication also remains constant. Concretely, we verified that if $n_T \oplus \mathsf{uid} \oplus K \oplus$ 'feedback bits' remains constant, then the keystream generated after authentication is constant as well. Here the 'feedback bits' are computed according to $g(x)$. This suggests that the secret key K is the initial state of the LFSR. This also suggests that the keystream feedback loop from the output back to the LFSR present in the Hitag2 cipher is not present on CRYPTO1, which greatly simplified the analysis.

Proceeding to the next step in the authentication protocol, the reader nonce n_R is fed into the LFSR as well. Note that earlier bits of n_R already affect the encryption of the later bits of n_R. At this point, the initialization is complete and the input bit of the LFSR is no longer used. Figure 5 shows the initialization diagram for both reader and tag. The only difference is that the reader generates n_R and then computes and sends $n_R \oplus \mathsf{ks}_1$, while the tag receives $n_R \oplus \mathsf{ks}_1$ and then computes n_R.

Note that we can, by selecting an appropriate key K, uid, and tag nonce n_T, totally control the state of the LFSR just before feeding in the reader nonce. In practice, if we want to observe the behavior of the LFSR starting in state α, we often set the key to 0, let the Ghost select a uid of 0 and compute which n_T we should let the Ghost send to reach the state α. Now, because n_T is only 32 bits long and α is 48 bits long, this does not seem to allow us to control the leftmost 16 bits of α: they will always be 0. In practice, however, many readers accept and process tag nonces of arbitrary length. So by sending an appropriate 48 bit tag nonce n_T, we can fully control the state of the LFSR just before the reader nonce. This will be very useful in the next section, where we describe how we recovered the filter function f.

5.2 Filter function

The first time the filter function f is used, is when the first bit of the reader nonce, $n_{R,0}$, is transmitted. At this point, we fully control the state α of the LFSR by setting the uid, the key, and the tag nonce. As before, we use the Ghost to send a uid of 0, use the key 0 on the reader, and use 48 bit tag nonces to set the LFSR state. So, for values α of our choice, we can observe $n_{R,0} \oplus f(\alpha)$, since that is what is being sent by the reader. Since we power up the reader every time, the generated reader nonce is the same every time. Therefore, even though we do not know $n_{R,0}$, it is a constant.

The first task is now to determine which bits of the LFSR are inputs to the filter function f. For this, we pick a random state α and observe $n_{R,0} \oplus f(\alpha)$. We then vary a single bit in α, say the ith, giving state α', and observe $n_{R,0} \oplus f(\alpha')$. If $f(\alpha) \neq f(\alpha')$, then the ith bit must be input to f. If $f(\alpha) = f(\alpha')$, then we can draw no conclusion about the ith bit, but if this happens for many choices of α, it is likely that the ith bit is not an input to f.

Figure 6 shows an example. The key in the reader (for block 0) is set to 0 and the Ghost sends a uid of 0. On the left hand side, the Ghost sends the tag nonce `0x6dc413abd0f3` and on the right hand side it sends the tag nonce

Sender	Hex	Hex	
Reader	26	26	req type A
Ghost	04 00	04 00	answer req
Reader	93 20	93 20	select
Ghost	00 00 00 00 00	00 00 00 00 00	uid,bcc
Reader	93 70 00 00 00 00 00 9c d9	93 70 00 00 00 00 00 9c d9	select(uid)
Ghost	08 b6 dd	08 b6 dd	MIFARE 1k
Reader	60 00 f5 7b	60 00 F5 7B	auth(block 0)
Ghost	6d c4 13 ab d0 f3	6d c4 13 ab d0 73	n_T
Reader	df 19 d5 7a e5 81 ce cb	5e ef 51 1e 5e fb a6 21	$n_R \oplus \mathsf{ks}_1, \mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$

Fig. 6. Nearly equal LFSR states

0x6dc413abd073. This leads, respectively, to LFSR states of 0xb05d53bfdb10 and 0xb05d53bfdb11. These differ only in the rightmost bit, i.e., bit 47. On the left hand side, the first bit of the encrypted reader nonce is 1 and on the right hand side it is 0 (recall the byte-swapping convention used in traces). Hence, bit 47 must be an input to the filter function f.

This way, we were able to see that the bits $9, 11, \ldots, 45, 47$ are input to the filter function f. Based on the similarity with the Hitag2, we guessed that there are 5 "first layer circuits" each taking four inputs, respectively, $9, 11, 13, 15$ for the left-most circuit up to $41, 43, 45, 47$ for the right-most circuit. The five results from these circuit are then, we guessed, input into a "second layer circuit", producing a keystream bit. (See Figure 8 for the structure of CRYPTO1). Note that in the Hitag2, all these circuits are "balanced", in the sense that for half the possible (16 or 32) inputs they give a 0 and for half the possible inputs they give a 1.

To verify our guess and to determine f, we again take a random state α of the LFSR. We then vary 4 (guessed) inputs to a first layer circuit in all 16 ways possible, giving states $\alpha_0, \alpha_1, \ldots \alpha_{15}$ and observe $r_0 \oplus f(\alpha_0), \ldots, r_0 \oplus f(\alpha_{15})$. If our guess was correct, we expect these to be 16 zeros, 16 ones, or 8 zeros and 8 ones: either the 16 non-varying inputs are such that the 4 varying inputs do not influence the keystream bit (in which case we get all zeros or all ones), or we get a "balanced" result as in the Hitag2. In the first two cases, we try again; in the latter case, we have found the component (up to a NOT, but that is irrelevant). Figure 7 shows examples of LFSRs that vary the inputs to a first layer circuit.

It turned out that our guess was correct; there are two different circuits used in the first layer. Two circuits in the first layer compute $f_a(x_3, x_2, x_1, x_0)$ represented by the boolean table 0x26c7 and the other three compute $f_b(x_3, x_2, x_1, x_0)$

LFSR \ XX	55	54	51	50	45	44	41	40	15	14	11	10	05	04	01	00
0xb05d53bfdbXX	0	0	0	0	1	1	0	1	1	1	0	1	0	0	1	1
0xfbb57bbc7fXX	1	1	1	1	0	0	1	0	0	0	1	0	1	1	0	0
0xe2fd86e299XX	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 7. First bit of encrypted reader nonce

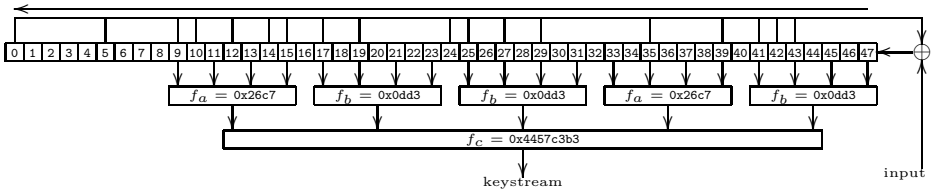

Fig. 8. The CRYPTO1 Cipher

represented by the boolean table `0x0dd3`. I.e., from left to right the bits of `0x26c7` are the values of $f_a(1,1,1,1), f_a(1,1,1,0), \ldots, f_a(0,0,0,0)$ and similarly for f_b (and f_c below). These five output bits are input into the circuit in the second layer. By trying 32 states that produce all 32 possible outputs for the first layer, we build a table for the circuits in the second layer. It computes $f_c(x_4, x_3, x_2, x_1, x_0)$ represented by the boolean table `0x4457c3b3`. In this way we recovered the filter function f. See Figure 8.

6 MIFARE Weaknesses and Exploits

This section describes four design flaws of the MIFARE Classic. These flaws allow us to recover the secret key from a genuine MIFARE reader in two different ways. In one way, the core of which is described in Section 6.1, we first have to gather a modest amount of data from the reader. Together with a precomputed table this can be used to invert the filter function f and then, with an LFSR rollback technique described in Section 6.2, we can recover the secret key. In the other way, described in Section 6.3, we can directly invert the filter function f in under one second on ordinary hardware without the need for any precomputed tables. The same LFSR rollback technique then also recovers the secret key. In Section 6.4 we finish with a weakness in the way that parity bits are treated.

6.1 LFSR State Recovery

The tag nonce directly manipulates the internal state of the LFSR. This enables us to recover the state of the LFSR, given a segment of keystream.

First, we build a table consisting of tuples $(\mathsf{lfsr}, \mathsf{ks})$ where lfsr runs over all LFSR states of the form `0x000WWWWWWWWW` and ks are the first 64 bits of keystream they generate. This one time computation can be performed on a ordinary computer and can be reused for any reader/key. This produces a table of 2^{36} rows.

Now we focus on a specific reader that we want to attack. For each 12 bit number `0xXXX`, we start an authentication session using the same uid. We set the challenge nonce of the tag to $n_T =$ `0x0000XXX0`. After the reader answers with $n_R \oplus \mathsf{ks}_1, \mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$ we do not reply. Then most readers send $\mathsf{halt} \oplus \mathsf{ks}_3$. Since we know $\mathsf{suc}^2(n_T)$ and halt we can recover $\mathsf{ks}_2, \mathsf{ks}_3$. There is exactly one value for `0xXXX` that produces an LFSR state of the form `0xYYYYYYYY000Y` after feeding in $n_T =$ `0x0000XXX0`. While feeding in the reader nonce n_R, the zeros in the LFSR

are shifted to the left, producing an LFSR state of the form 0x000YZZZZZZZZ. Since we have all LFSR states of this form in our table, we can recover it by searching for $\mathsf{ks}_2, \mathsf{ks}_3$.

Typically, only for a single value of 0xXXX do we get a hit in our table, because the size of the keystream is 64 bits and the size of the LFSR is only 48 bits. In Section 6.2 we show how we can use the LFSR state that we find in the table, together with n_T and $n_R \oplus \mathsf{ks}_1$, to obtain the secret key.

In the above description it is possible to trade off between the size of the lookup table and the number of authentication sessions needed. In the above setup, the size of the table is approximately one terabyte and the number of required authentication sessions is 4096. For instance, by varying 13 instead of 12 bits of the tag nonce we halve the size of the table at the cost of doubling the number of required sessions.

Note that even if the reader does not respond in case of time out, we can still use this technique to recover the LFSR state. In that case, for each 0xXXX, we search only for the corresponding ks_2 in the table. Since there are 2^{48-12} entries in the table, and ks_2 is 32 bits long, we get on average 2^4 matches. Since we are considering 2^{12} possible values of 0xXXX, we get a total of approximately 2^{16} possible LFSR states. Each of these LFSR states gives us, using Section 6.2, a candidate key. With a single other partial authentication session, i.e., one up to and including the answer from the reader, we can then check which of those keys is the correct one.

6.2 LFSR Rollback

Given the state $r_k r_{k+1} \dots r_{k+47}$ of the LFSR at a certain time k (and the input bit, if any), one can use the relation (1) to compute the previous state $r_{k-1} r_k \dots r_{k+46}$.

Now suppose that we somehow learned the state of the LFSR right after the reader nonce has been fed in, for instance using the approach from the previous section, and that we have eavesdropped the encrypted reader nonce. Because we do not know the plaintext reader nonce, we cannot immediately roll back the LFSR to the state before feeding in the reader nonce. However, the input to the filter function f does not include the leftmost bit of the LFSR. This weakness does enable us to recover this state (and the plaintext reader nonce) anyway.

To do so we shift the LFSR to the right; the rightmost bit falls out and we set the leftmost bit to an arbitrary value r. Then we compute the function f and we get one bit of keystream that was used to encrypt the last bit $n_{R,31}$ of the reader nonce. Note that the leftmost bit of the LFSR is not an input to the function f, and therefore our choice of r is irrelevant. Using the encrypted reader nonce we recover $n_{R,31}$. Computing the feedback of the LFSR we can now set the bit r to the correct value, i.e., so that the LFSR is in the state prior to feeding $n_{R,31}$. Repeating this procedure 31 times more, we recover the state of the LFSR before the reader nonce was fed in.

Since the tag nonce and uid are sent as plaintext, we also recover the LFSR state before feeding in $n_T \oplus \mathsf{uid}$ (step 4). Note that this LFSR state is the secret key!

6.3 Odd Inputs to the Filter Function

The inputs to the filter function f are only on odd-numbered places. The fact that they are so evenly placed can be exploited. Given a part of keystream, we can generate those relevant bits of the LFSR state that give the even bits of the keystream and those relevant bits of the LFSR state that give the odd bits of the keystream separately. By splitting the feedback in two parts as well, we can combine those even and odd parts efficiently and recover exactly those states of the LFSR that produce a given keystream. This may be understood as "inverting" the filter function f.

Let $b_0 b_1 \dots b_{n-1}$ be n consecutive bits of keystream. For simplicity of the presentation we assume that n is even; in practice n is either 32 or 64. Our goal is to recover all states of the LFSR that produce this keystream. To be precise, we will search for all sequences $\bar{r} = r_0 r_1 \dots r_{46+n}$ of bits such that

$$\begin{aligned} r_k \oplus r_{k+5} \oplus r_{k+9} \oplus r_{k+10} \oplus r_{k+12} \oplus r_{k+14} \oplus r_{k+15} \oplus r_{k+17} \\ \oplus r_{k+19} \oplus r_{k+24} \oplus r_{k+25} \oplus r_{k+27} \oplus r_{k+29} \oplus r_{k+35} \oplus r_{k+39} \oplus r_{k+41} \\ \oplus r_{k+42} \oplus r_{k+43} \oplus r_{k+48} = 0, \text{ for all } k \in \{0, \dots, n-2\}, \end{aligned} \quad (2)$$

and such that

$$f(r_k \dots r_{k+47}) = b_k, \text{ for all } k \in \{0, \dots, n-1\}. \quad (3)$$

Condition (2) says that $\bar{r}$ is generated by the LFSR, i.e., that $r_0 r_1 \dots r_{47}$, $r_1 r_2 \dots r_{48}$, ... are successive states of the LFSR; Condition (3) says that it generates the required keystream. Since f only depends on 20 bits of the LFSR, we will overload notation and write $f(r_{k+9}, r_{k+11}, \dots, r_{k+45}, r_{k+47})$ for $f(r_k \dots r_{k+47})$. Note that when n is larger than 48, there is typically only one sequence satisfying (2) and (3), otherwise there are on average 2^{48-n} such sequences.

During our attack we build two tables of approximately 2^{19} elements. These tables contain respectively the even numbered bits and the odd numbered bits of the LFSR sequences that produce the evenly and oddly numbered bits of the required keystream.

We proceed as follows. Looking at the first bit of the keystream, b_0, we generate all sequences of 20 bits $s_0 s_1 \dots s_{19}$ such that $f(s_0, s_1, \dots, s_{19}) = b_0$. The structure of f guarantees that there are exactly 2^{19} of these sequences. Note that the sequences $\bar{r}$ of the LFSR that we are looking for must have one of these sequences as its bits $r_9, r_{11}, \dots, r_{47}$.

For each of the entries in the table, we now do the following. We view the entry as the bits $9, 11, \dots, 47$ of the LFSR. We now shift the LFSR two positions to the left. The feedback bit, which we call s_{20}, that is shifted in second could be either 0 or 1; not knowing the even numbered bits of the LFSR nor the low numbered odd ones, we have no information about the feedback. We can check, however, which of the two possibilities for s_{20} matches with the keystream, i.e., which satisfy $f(s_1, s_2, \dots, s_{20}) = b_2$. If only a single value of s_{20} matches, we extend the entry in our table by s_{20}. If both match, we duplicate the entry,

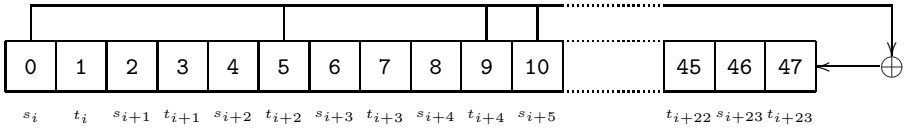

Fig. 9. Subsequences $\bar{s}$ and $\bar{t}$

extending it once with 0 and once with 1. If neither matches, we delete the entry. On average, 1/4 of the time we duplicate an entry, 1/4 of the time we delete an entry, and 1/2 of the time we only extend the entry. Therefore, the table stays, approximately, of size 2^{19}.

We repeat this procedure for the bits $b_4, b_6, \ldots, b_{n-1}$ of the keystream. This way we obtain a table of approximately 2^{19} entries $s_0 s_1 \ldots s_{19+n/2}$ with the property that $f(s_i, s_{i+1}, \ldots, s_{i+19}) = b_{2i}$ for all $i \in \{0, 1, \ldots, n/2\}$. Consequently, the sequences $\bar{r}$ of the LFSR that we are looking for must have one of the entries of this table as its bits $r_9, r_{11}, \ldots, r_{47+n}$.

Similarly, we obtain a table of approximately 2^{19} entries $t_0 t_1 \ldots t_{19+n/2}$ with the property that $f(t_i, t_{i+1}, \ldots, t_{i+19}) = b_{2i+1}$ for all $i \in \{0, 1, \ldots, n/2\}$.

Note that after only 4 extensions of each table, when all entries have length 24, one could try every entry $s_0 s_1 \ldots s_{23}$ in the first table with every entry $t_0 t_1 \ldots t_{23}$ in the second table to see if $s_0 t_0 s_1 \ldots t_{23}$ generates the correct keystream. Note that this already reduces the search complexity from 2^{48} in the brute force case to $(2^{19})^2 = 2^{38}$.

To further reduce the search complexity, we now look at the feedback of the LFSR. Consider an entry $\bar{s} = s_0 s_1 \ldots s_{19+n/2}$ of the first table and an entry $\bar{t} = t_0 t_1 \ldots t_{19+n/2}$ of the second table. In order that $\bar{r} = s_0 t_0 s_1 \ldots t_{19+n/2}$ is indeed generated by the LFSR, it is necessary (and sufficient) that every 49 consecutive bits satisfy the LFSR relation (2), i.e., the 49th must be the feedback generated by the previous 48 bits.

So, for every subsequence $s_i s_{i+1} \ldots s_{i+24}$ of 25 consecutive bits of $\bar{s}$ we compute its contribution $b_i^{1,\bar{s}} = s_k \oplus s_{i+5} \oplus s_{i+6} \oplus s_{i+7} \oplus s_{i+12} \oplus s_{i+21} \oplus s_{i+24}$ of the LFSR relation and for every subsequence $t_i t_{i+1} \ldots t_{i+23}$ of 24 consecutive bits of $\bar{t}$ we compute $b_i^{2,\bar{t}} = t_{i+2} \oplus t_{i+4} \oplus t_{i+7} \oplus t_{i+8} \oplus t_{i+9} \oplus t_{i+12} \oplus t_{i+13} \oplus t_{i+14} \oplus t_{i+17} \oplus t_{i+19} \oplus t_{i+20} \oplus t_{i+21}$. See Figure 9. If $s_0 t_0 s_1 \ldots t_{n/2}$ is indeed generated by the LFSR, then

$$b_i^{1,\bar{s}} = b_i^{2,\bar{t}} \text{ for all } i \in \{0, \ldots, n/2 - 5\}. \tag{4}$$

Symmetrically, for every subsequence of 24 consecutive bits of $\bar{s}$ and corresponding 25 consecutive bits of $\bar{t}$, we compute $\tilde{b}_i^{1,\bar{s}} = s_{i+2} \oplus s_{i+4} \oplus s_{i+7} \oplus s_{i+8} \oplus s_{i+9} \oplus s_{i+12} \oplus s_{i+13} \oplus s_{i+14} \oplus s_{i+17} \oplus s_{i+19} \oplus s_{i+20} \oplus s_{i+21}$ and $\tilde{b}_i^{2,\bar{t}} = t_i \oplus t_{i+5} \oplus t_{i+6} \oplus t_{i+7} \oplus t_{i+12} \oplus t_{i+21} \oplus t_{i+24}$. Also here, if $s_0 t_0 s_1 \ldots t_{n/2}$ is indeed generated by the LFSR, then

$$\tilde{b}_i^{1,\bar{s}} = \tilde{b}^{2,\bar{t}} \text{ for all } i \in \{0, \ldots, n/2 - 5\}. \tag{5}$$

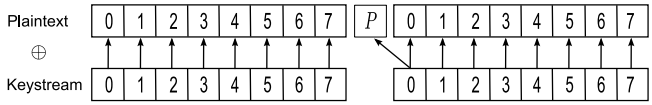

Fig. 10. Encryption of parity bits

One readily sees that together, conditions (4) and (5) are equivalent to equation (2).

To efficiently determine the LFSR state sequences that we are looking for, we sort the first table by the newly computed bits $b_0^{1,\bar{s}} \dots b_{n/2-5}^{1,\bar{s}} \tilde{b}_0^{1,\bar{s}} \dots \tilde{b}_{n/2-5}^{1,\bar{s}}$, and the second table by $b_0^{2,\bar{t}} \dots b_{n/2-5}^{2,\bar{t}} \tilde{b}_0^{2,\bar{t}} \dots \tilde{b}_{n/2-5}^{2,\bar{t}}$.

Since $s_0 t_0 s_1 \dots t_{n/2}$ is generated by the LFSR if and only $b^{1,\bar{s}} \tilde{b}^{1,\bar{s}} = b^{2,\bar{t}} \tilde{b}^{2,\bar{t}}$ and since by construction it generates the required keystream, we do not even have to search anymore. The complexity now reduces to n loops over two tables of size approximately 2^{19} and two sortings of these two tables. For completeness sake, note that from our tables we retrieve $r_9 r_{10} \dots r_{46+n}$. So to obtain the state of the LFSR at the start of the keystream, we have to roll back the state $r_9 r_{10} \dots r_{58}$ 9 steps.

In a variant of this method, applicable if we have sufficiently many bits of keystream available (64 will do), we only generate one of the two tables. For each of the approximately 2^{19} entries of the table, the LFSR relation (1) can then be used to express the 'missing' bits as linear combinations (over $\mathbb{F}_2$) of the bits of the entry. We can then check if it produces the required keystream.

This construction has been implemented in two ways. First of all as C code that recovers states from keystreams. Secondly also as a logical theory that has been verified in the theorem prover PVS [ORSH95]. The latter involves a logical formalization of many aspects of the MIFARE Classic [JW08].

6.4 Parity Bits

Every 8 bits, the communication protocol sends a parity bit. It turns out that the parity is not computed over the ciphertext, at the lowest level of the protocol, but over the plaintext. The parity bits themselves are encrypted as well; however, they are encrypted with the same bit of keystream that is used to encrypt the next bit. Figure 10 illustrates the mapping of the keystream bits to the plaintext.

In general, this leaks one bit of information about the plaintext for every byte sent. This can be used to to drastically reduce the search space for tag nonces in Section 8.

7 Attacking MIFARE

Attack One. Summarizing, an attacker can recover the secret key from a MIFARE reader as follows.

First, the attacker generates the table of $(\mathsf{lfsr}, \mathsf{ks})$ tuples as described in Section 6.1. This one terabyte table can be computed in one afternoon on standard hardware and can be reused.

Next, the attacker initiates $4096 = 2^{12}$ authentication sessions and computes $\mathsf{ks}_2, \mathsf{ks}_3$ for each of these sessions as described in Section 4.1. Note that this only requires access to a reader and not to a tag. As explained in Section 6.1, it is possible to recover the state of the LFSR prior to feeding in n_R. Then, as explained in Section 6.2, it is also possible to recover the state prior to feeding in $n_T \oplus \mathsf{uid}$. I.e., the secret key is recovered!

Experiments show that it is typically possible to gather between 5 and 35 partial authentication sessions per second from a MIFARE reader, depending on whether or not the reader is online. This means that gathering 4096 sessions takes between 2 and 14 minutes.

Attack Two. Instead of using the table, we can also use the invertibility of f described in Section 6.3 to recover the state of the LFSR at the end of the authentication. This way, we only need a single (partial) authentication session.

Note that this attack cannot be stopped by fixing the readers to not continue communication after communication fails. With the knowledge of just ks_2, we can invert f to find approximately 65536 candidate keys; these can be checked against another authentication session.

In practice, a relatively straightforward implementation of this attack takes less than one second of computation and only about 8 MB of memory on ordinary hardware to recover the secret key. Moreover, it does not require any kind of pre-computation, rainbow tables, etc. A highly optimized implementation of the single table variant consumes virtually no memory and recovers the secret key within 0.1 second on the same hardware.

8 Multiple-Sector Authentication

Many systems authenticate for more than one sector. Starting with the second authentication the protocol is slightly different. Since there is already a session key established, the new authentication command is sent encrypted with this key. At this stage the secret key K' for the new sector is loaded into the LFSR. The difference is that now the tag nonce n_T is sent encrypted with K' while it is fed into the LFSR (resembling the way the reader nonce is fed in). From this point on the protocol continues exactly as before, i.e., the reader nonce is fed in, etc.

To clone a card, one typically needs to recover all the information read by the reader and this usually involves a few sectors. To do so, we first eavesdrop a single, complete session which contains authentications for multiple sectors. Once we have recovered the key for the first sector as described in Section 7, we proceed to the next sector read by the reader. The authentication request is now encrypted with the previous session key, but this is not a problem: we just recovered that key, so we can decrypt the authentication request. The issue

now is that we need the tag nonce n_T to mount our attacks and it is encrypted with the key K' which we do not yet know. We can, of course, simply try all 2^{16} possible tag nonces to execute our attack.

Using the parity bits, however, the number of possible tag nonces can be drastically reduced. The first three parity bits, say p_0, p_1, p_2, of the tag nonce n_T are encrypted with the keystream bits that are also used to encrypt bits n_8, n_{16}, and n_{24} of n_T. That is, from the communication we can observe $p_0 \oplus b_8$, $n_8 \oplus b_8$, where b_8 is the keystream bit that is used to encrypt n_8, and similarly for the other two parity bits. From this we can see whether or not p_0, the parity of the first byte of n_T, is equal to n_8, the first bit of the second byte of n_T. This information decreases the number of potential nonces by a factor of 2. The same holds for the other 2 parity bits in n_T and for the 7 parity bits in $\mathsf{suc}^2(n_T)$ and $\mathsf{suc}^3(n_T)$. In total, the search space is reduced from 2^{16} nonces to only $2^{16}/2^{10} = 64$ nonces.

A not yet well-understood phenomenon allows us to select almost immediately the correct nonce out of those 64 candidates. The pseudo-random generator of the tag keeps shifting during the communication in a predictable way. This enables us the predict the distance $d(n_T, n'_T)$ between the tag nonce n_T used in one authentication session and the tag nonce n'_T used in the next. Distance here means the number of times the pseudo-random number generator has to shift after outputting n_T before it outputs n'_T. The relation we found experimentally is $d(n_T, n'_T) = 8t - 55c - 400$, where t is the time between the sending of the encrypted reader nonce in the first authentication session and the authenticate command that starts the next session (expressed in bit-periods, the time it takes to send a single bit, approximately $9.44\mu s$) and c is the number of commands the reader sends in the first session. However, we do not know precisely why this relation holds and if it holds under all circumstances. In practice, the correct nonce is nearly always the one (from the 64 candidates) whose distance to n_T is closest to $d(n_T, n'_T)$. Consequently, keys for subsequent sectors are obtained at the same speed as the key for the first sector.

9 Consequences and Conclusions

We have reverse engineered the security mechanisms of the MIFARE Classic chip. We found several vulnerabilities and successfully managed to exploit them, retrieving the secret key from a genuine reader. We have presented two very practical attacks that, to retrieve the secret key, do not require access to a genuine tag at any point.

In particular, the second attack recovers a secret key from just one or two authentication attempts with a genuine reader (without access to a genuine tag) in less than a second on ordinary hardware and without any pre-computation. Furthermore, an attacker that is capable of eavesdropping the communication between a tag and a reader can recover all keys used in this communication. This enables an attacker to decrypt the whole trace and clone the tag.

What the actual implications are for real life systems deploying the MIFARE Classic depends, of course, on the system as a whole: contactless smart cards are generally not the only security mechanism in place. For instance, public transport payment systems such as the Oyster card and OV-Chipkaart have a back-end system recording transactions and attempting to detect fraudulent activities (such as traveling on a cloned card). Systems like these will now have to deal with the fact that it turns out to be fairly easy to read and clone cards. Whether or not the current implementations of these back ends are up to the task should be the subject to further scrutiny. We would also like to point out that some potential of the MIFARE Classic is not being used in practice, viz., the possibility to use counters that can only be decremented, and the possibility to read random sectors for authentication. Whether or not this is sufficient to salvage the MIFARE Classic for use in payment systems is the subject of further research [TN08].

In general, we believe that it is far better to use well-established and well-reviewed cryptographic primitives and protocols than proprietary ones. As was already formulated by Auguste Kerckhoffs in 1883, and what is now known as Kerckhoffs' Principle, the security of a cryptographic system should not depend on the secrecy of the system itself, but only on the secrecy of the key [Ker83]. Time and time again it is proven that details of the system will eventually become public; the previous obscurity then only leads to a less well-vetted system that is prone to mistakes.

Acknowledgements

The authors are thankful to Peter Dolron and all the people from TechnoCentrum (TeCe, FNWI) for developing the hardware for the Ghost. We also are indebted to Wouter Teepe for tirelessly dealing with all the subtle political issues surrounding this topic and to Jaap-Henk Hoepman for starting it all and for his advice in the early stages of this project. Finally, we would like to thank Ravindra Kali and Vinesh Kali for running many errands during the intense time we worked on this project.

References

[HHJ+06] Hoepman, J.-H., Hubbers, E., Jacobs, B., Oostdijk, M., Wichers Schreur, R.: Crossing borders: Security and privacy issues of the European e-passport. In: Yoshiura, H., Sakurai, K., Rannenberg, K., Murayama, Y., Kawamura, S.-i. (eds.) IWSEC 2006. LNCS, vol. 4266, pp. 152–167. Springer, Heidelberg (2006)

[ISO01] ISO/IEC 14443. Identification cards - Contactless integrated circuit(s) cards - Proximity cards (2001)

[JW08] Jacobs, B., Wichers Schreur, R.: Mifare Classic, logical formalization and analysis, PVS code (manuscript, 2008)

[Ker83] Kerckhoffs, A.: La cryptographie militaire. Journal des Sciences Militaires IX, 5–38 (1883)

[KHG08] de Koning Gans, G., Hoepman, J.-H., Garcia, F.D.: A practical attack on the MIFARE Classic. In: Proceedings of the 8th Smart Card Research and Advanced Application Workshop (CARDIS 2008). LNCS, vol. 5189, pp. 267–282. Springer, Heidelberg (2008)

[NESP08] Nohl, K., Evans, D., Starbug, Plötz, H.: Reverse-engineering a cryptographic RFID tag. In: USENIX Security 2008 (2008)

[NP07] Nohl, K., Plötz, H.: Mifare, little security, despite obscurity. In: Presentation on the 24th Congress of the Chaos Computer Club. Berlin (December 2007)

[ORSH95] Owre, S., Rushby, J.M., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering 21(2), 107–125 (1995)

[TN08] Teepe, W., Nohl, K.: Making the best of MIFARE Classic (manuscript, 2008)