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.

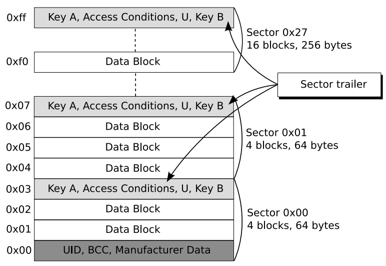


Fig. 1. Logical structure

4 Authentication Protocol

当tag进入加电的reader的电磁场的时候,

... r and powers up, it

它直接通过发送它的UID开始防冲突协议

... uid. The reader then selects this tag as specified in ISO14443-A [ISO01] ...

根据厂商的文档记录,reader接着发送指定块的认证请求

reader接着选择这个tag

然后,tag获取一个challenge随机数nt并且直接发送它到reader.

接着,reader发送它自己的challenge随机数nr和ar的应答一起到tag的challenge.

... ag. The tag finishes

tag最终完成认证,通过回复at到reader的challenge.

... Starting with n_R,

从nr开始,所有的通信都被加密了.这意味着nr,ar和at都被异或了,用密钥流ks1,ks2,ks3.图2显示了一个样例.

... XOR-ed

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	$n_R \oplus ks_1, a_R \oplus ks_2$
10	Tag	8b d4 10 08	$a_T \oplus ks_3$

Fig. 2. Authentication Trace

我们开始我们控制的OpenPCD reader和Ghost进行试验

control. T tag的伪随机数生成器是完全确定的 ic. Therefore

因此,它生成的随机数仅仅依赖于在加电和开始通信之间的时间 start

一旦我们控制了reader,我们就控制了时序,因此每次都能够得到相同的tag随机数.

把Ghost操作当作一个tag,我们能够选择自定义的challenge随机数和uid Ghost operating as a

ermore, by fixing n_T

此外,通过固定nt(和uid)并且重复认证,我们发现reader每次重新启动的时候生成相同序列的随机数

the same sequence of nonces every time it is restarted. Unlike in the tag, the

不像在tag中,reader中伪随机数生成器的状态并不在每个时钟tick时进行更新而是在每次调用的时候.

在tag中的伪随机数生成器用于生成nt的是一个16位LFSR,使用生成多项式x^16...

由于随机数是32位长,而LFSR有16位状态 ng

and the LFSR has nt的前一半决定了后一半.

这意味着给定一个32位数,我们能够告诉你它是否是一个合法的tag随机数.

如果它是由LFSR生成的,准确的说,一个32位数n0n1...n31是一个合法的tag随机数,当且仅当nk@nk+2@nk+3@nk+5@nk+16=0

注意Ghost能够发送任意的值作为随机数,而不仅局限于合法的tag随机数.

我们注意到如果nt@uid保持不变,那么加密的reader的challenge随机数也保持不变

然而,在两次会话中的应答at和ar有不同的密文. he answers a_T and a_R, however,

For example, in Figure 2, the

例如,在图2中uid是0xc2a82df4,nt是0x4297c0a4,因此nt@uid是0x803fed50.

如果我们替换uid为0x1dfbe033,nt为0x9dc40d63,那么nt@uid仍然等于0x803fed50.

在所有的案例中,加密的reader随机数nr@ks1都是0x7ddb9b83

然而,在图2中的ar@ks2是0x67eb5d83,at@ks2是0x8bd41008.

当uid和nt被修改之后,它们分别是0x4295c446和0xeb3ef7da. respectively,

这表明它们运行的密钥流是相同的,也表明at和ar依赖nt.

that c 通过异或所有的应答ar@ks2和ar`@ks2,我们可以得到ar@ar`.

我们注意到ar@ar`是一个合法的tag随机数. Because

因为合法的tag随机数集合是F32x2的线性子空间.当两个元素属于F2时,合法的tag随机数的异或也会是一个合法的tag随机数.这表明ar和ar`都是合法的tag随机数.

给定一个通过LFSR生成的32位随机数nt,可以计算后继suc(nt),由后面的32个生成位组成.

在这个阶段,我们校验ar@ar`=suc2(nt@nt`)=suc2(nt)@suc2(nt`),猜想ar=suc2(nt),ar`=suc2(nt`),类似的,tag的应答,我们校验得到at=suc3(nt)以及at`=suc3(nt`).

总结,认证协议可以如下描述,如图3. llows; 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

在tag发送随机数nt之后,tag和reader都使用共享的key K,uid和随机数nt来初始化cipher.接着reader取它的challenge随机数nr并且用密钥流ks1的第一部分加密发送.接着它用nr更新cipher的状态.reader认证通过发送加密的suc2(nt).

	Tag		Reader
0		anti-c(uid) $\rightarrow$	
1		auth(block) $\leftarrow$	
2	picks n_T		
3		n_T $\rightarrow$	
4	$\mathsf{ks}_1 \leftarrow \mathsf{cipher}(K, \mathsf{uid}, n_T)$		$\mathsf{ks}_1 \leftarrow \mathsf{cipher}(K, \mathsf{uid}, n_T)$
5			picks n_R
6			$\mathsf{ks}_2, \ldots \leftarrow \mathsf{cipher}(n_R)$
7		$n_R \oplus \mathsf{ks}_1, \mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$ $\leftarrow$	
8	$\mathsf{ks}_2, \ldots \leftarrow \mathsf{cipher}(n_R)$		
9		$\mathsf{suc}^3(n_T) \oplus \mathsf{ks}_3$ $\rightarrow$	

Fig. 3. Authentication Protocol

在这个点上,tag可能用同样的方法更新cipher的状态,并且校验reader的认证.

密钥流的其他部分ks3,ks4...现在被确定了,并且现在所有的通信都被加密了.即异或的密钥流.tag完成认证协议通过发送suc3(nt)@ks3.现在reader可以校验tag的认证.

reader is able to verify the authenticity of the tag.

4.1 Known Plaintext

从认证协议的描述中很容易看出部分的keystream能够被恢复.

From the description of the authentication protocol it is easy to see that parts

如果知道了nt和suc2(nt)@ks2,就能够恢复ks2.即32位的keystream,通过计算suc2(nt)并且异或.

此外,实验证明,如果认证协议的第9步,tag不发送任何信息,接着大多数reader将超时并且发送一个HALT命令.由于通信被加密了,它通常发送HALT@ks3.

halt command. Since communication is encrypted it actually sends halt $\oplus$ ks_3.

已知HALT命令的字节码(0x500057cd),我们就能够恢复ks3.

如果一些reader不发送HALT命令,而是继续,如果认证成功的话

这通常意味着它发送一个加密的读命令,由于读命令的字节码也是已知的(KHG08),因此这也能够让我们恢复ks3,通过猜测块号.

It is import… rather, a parti… from a reader… without using…

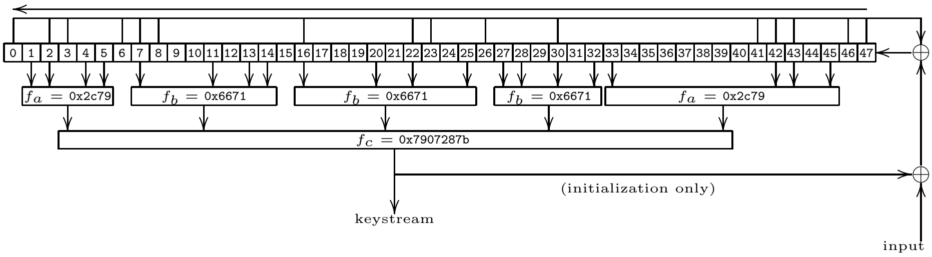
需要注意的是,我们能够得到这样的一个认证对话(或者相反,部分的认证对话,作为Ghost从不认证自身),来自一个reader,不知道的安全密钥,实际上,不使用tag.???

如果一个攻击能够访问所有的tag和reader.并且能够窃听一个成功(完整的)认证对话,那么所有的ks2和ks3从tag和reader的应答suc2(nt)@ks2和suc3(nt)@ks3能够被恢复,它甚至可以工作在reader在超时之后不发送HALT或读命令.

5 CRYPTO1 Cipher

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

CRYPTO01算法的核心是一个48位线性反馈左移寄存器(LFSR),生成多项式为g(x)=.....



这个多项式在NESP08中给出.

注意它也能够从相关的uid和在NP07中描述的安全密钥中被推导出来.

$x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^9 + x^7 + x^6 + x^5 + 1$. 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

在每个时钟tick,寄存器左移一位. 最左边的位被丢弃,并且反馈位根据g(x)计算出来.

附加的,LFSR有一个输入位,和反馈位异或后,从右边加载进入LFSR. that is To be

准确的说,如果LFSR在时间k的状态是rkrk+1...rk+47,并且输入位是i,那么它在时间k+1的状态是rk+1rk+2..rk+48,其中.

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

The input bit i is only used during initialization. 输入位i仅仅在初始化期间使用.

为了加密,LFSR选择的位被放入到一个过滤函数f通过. f.

实际上LFSR的哪个位被放入通过f,并且f是什么,并没有在NP07中说明.

was not revealed in [NP07]. 注意CRYPTO1的生成结构非常类似于Hitag2.

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

有一个来自NXP的低频tag,在Hitag2使用的cipher的描述在互联网上是自由的.

我们使用这来建立关于cipher的初始化的描述细节猜想(看在下面Section 5.1),以及过滤函数f的细节描述(看下面的Section 5.2)

5.1 Initialization

在认证协议期间LFSR被初始化.

一如既往,我们尝试使用不同的参数运行几个认证对话,正如我们在Section 4中提到的,

如果nt@uid保持不变,那么加密的reader随机数也保持不变

stant. This suggests that $n_T \oplus$ uid is first fed into the LFSR. Moreover, experiments showed that, if special care is taken with the

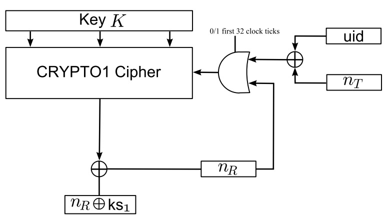


Fig. 5. Initialization Diagram

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

于是猜想nt@uid首先被加载进入LFSR.
此外,实验显示,如果对反馈位采取特别措施

以这样的方式修改nt@uid和安全密钥K.

feedback bits, it is possible to modify $n_T \oplus$ uid and the secret key K in such a way

那么在认证之后的密文也保持不变是可能的. …nstant. Concretely, we

具体而言,我们校验了如果nt@uid@K@feedback bit保持不变,那么认证之后生成的keystream也是不变的. …s constant, then the keystream

这儿,feedback bit由g(x)计算.

computed according to $g(x)$. T… 我们猜想,安全密钥K就是LFSR的初始状态.

这也让我们猜想keystream feedback循环从输出回到LFSR,只在Hitag2算法中存在,而在CRYPTO1中不存在.这大大简化了分析.

在认证协议处理到下一步,reader随机数nr加载进入LFSR,注意nr早期的位也影响nr的后期的加密位.

encryption of the later bits of n_R. At this point, the initialization is complete

此时,初始化完成,并且LFSR的输入位不再使用. …used. Figure 5 shows the initialization

图5显示了reader和tag的初始化流程图

它们仅仅不同在,reader生成nr并且接着计算发送nr@ks1,这时tag接收nr@ks1并且接着计算nr.

n_R and then computes and sends … then computes n_R.

我们需要注意的是,选择一个合适的密钥K,uid和tag随机数nt,完全控制了LFSR的状态,在reader随机数加载之前. … n_T, … In practice, …

事实上,如果我们想观察LFSR在状态a的行为,我们通常设置密钥为0,让Ghost选择uid为0,并且计算nt,我们将让Ghost发送到达的状态a.

often set …

现在,由于nt仅仅是32位长,而a是48位长,这看起来似乎不允许我们控制a最左边的16位.它们通常是0.

事实上,然而,很多reader接受并且处理任意长度的tag随机数.

and process tag nonces of arbitrary length. So by sending an appropriate 48 bit

因此,通过发送任意48位tag随机数nt,我们能够完全控制在reader随机数之前的LFSR的状态,在下一section,这将是非常有用的,它将描述我们怎么恢复过滤函数f.

第一次过滤函数f被使用,是在reader随机数第一位nr,0被传输的时候.

The first time the filte… 此时,我们完全控制了LFSR的状态,通过设置uid,key,tag随机数.

nonce, $n_{R,0}$, is transmitted. At this point, we fully control the state α of the

之前,我们使用Ghost发送0值的uid,0值的key在reader上.并且使用48位tag随机数来设置LFSR状态.

Ghost to send a uid of 0, use the key 0 on the reader, and use 48 bit tag nonces

因此,我们选择的a值,我们能够观察到nr.0@f(a),因为它通过reader发送. …e $n_{R,0} \oplus f(\alpha)$,

since … every … thoug…

由于我们在任何时间加电reader而生成的reader随机数都是相同的.因此,即使我们不知道nr0,它是一个常数.

第一个任务是现在确定到过滤函数f的是LFSR输入的哪个位

对此,我们选取一个随机状态a和观察nr0@f(a),我们接着变换a中的一个位,叫第i位,得到状态a`,并且观察nr0@fa`,如果f(a)#f(a`),那么第i位必然是f的输入.如果f(a)=f(a`),那么我们关于第i位没有结论.但是如果有a的很多选择发生,它可能表明第i位不是f的输入.

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

图6显示了一个样例.在reader的key(block 0)被设置为0,Ghost发送0值uid,在左手边,Ghost发送的tag随机数为0x6dc413abd0f3,在右手边,它发送的tag随机数为

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	$n_R \oplus \mathsf{ks}_1, \mathsf{suc}^2(n_T) \oplus \mathsf{ks}_2$

这导致,分别对应0xb05d53bfdb10和0xb05d53bfdb11的LFSR状态

0x6dc413ab 它们仅仅在最右边位是不同的.即47位.

and 0xb05d53bfdb1 在左手边,加密的reader随机数首位为1,在右手边是0.

left hand si 回忆在traces中使用的byte-swapping convention he 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 因此,位47必然是过滤函数f的输入.

This way, 同样的方法,我们能够得到位9,11..47也是过滤函数f的输入.

基于Hitag2的相似性,我们猜想有5个'首层电路'每次取4个输入

分别为9,11,13,15为最左边电路,直到41,43,45,47为最右边电路.

le 这些电路的5个结果,我们猜想,输入到'第二层电路'产生一个keystream位

fr producing a keystream bit. (See Figure 8 for the structure of CRYPTO1). Note that 注意在Hitag2中,所有的电路是平衡的,从某种意义上说,一半是可能的输入,另一半是可能的.???

the p give

为了检验我们的猜想并确定f,我们又取一个随机的LFSR状态a,我们接着变换4个输入作为第一层电路,在所有16种可能的,得到状态a0,a1,...a15.观察r0@f(a0)...r0@f(a15).

如果我们的猜想是正确的,我们希望这是16个0,16个1,或者8个0和8个1

无论16个未变化的输入是怎样的,4个变换的输入不会影响keystream位(这种情况下我们得到全0或全1),或者我们得到一个平衡的结果象Hitag2一样.

在前两个案例中 事实表明我们的猜想是正确的.有两个不同的电路使用在第一层.两个电路在第一层计算fa(x3,x2,x1,x0),保留布尔表0x26c7,而其他三个计算fb(x3,x2,x1,x0) 但是不相关的)???

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

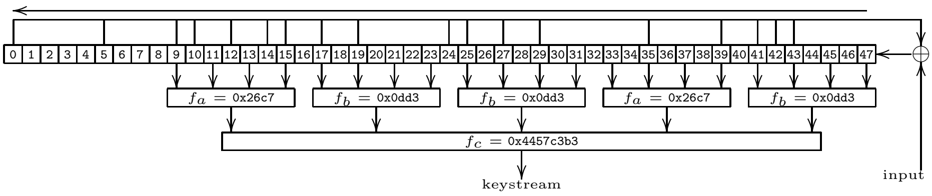


Fig. 8. The CRYPTO1 Cipher

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

这一section描述了4种设计缺陷.这些缺陷允许我们用两种不同的方法从一个真正的MIFARE reader恢复安全密钥

In one way, the core of which is described in Section 6.1, we first have to gather

一种方法,它的核心部分在section 6.1中描述,我们首先不得不从reader中收集适量的数据.通过预算表,我们能够用于反转过滤函数,然后使用在section 6.2中描述的LFSR回溯技术,我们能够恢复安全密钥.

techn... way, unde... table

另一种方法,在section 6.3中描述,我们能够直接反转过滤函数f,一秒内在一个ordinary硬件中,不再需要任何预算表.接着使用同样的LFSR回溯技术,我们能够恢复安全密钥.

Section 6.4 we finish with a weakness in the way that parity bits are treated.

在section 6.4我们完成了一个弱点,奇偶校验位是treated.??

6.1 LFSR State Recovery

tag随机数直接操纵LFSR的内部状态.这能够让我们恢复LFSR的状态.得到keystream的片段

首先,我们建立一个表,包含(lfsr,ks)对,lfsr运行覆盖格式如0x000WWWWWWWWW的所有LFSR状态.并且ks是它们生成的keystream的首64位.

they generate. This one time computation can be performed on a ordinary com-

这一次性的计算能够被执行在一个序列计算器上,并且能够被重复使用在任何reader/kev.这个表有2^36行.

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

现在我们专注于我们想攻击的特定reader上,每12位数0xXXX,我们使用相同的uid开始一个认证对话,我们设置tag的challenge随机数为nt=0x000XXX0,然后reader用nr@ks1,suc2(nt)@ks2回答,我们并不回复.接着很多reader发送halt@ks3.一旦我们知道了suc2(nt)和halt,我们就能够恢复ks2,ks3.

这是刚好一个值为0xXXX,产生格式如0xYYYYYYYY000Y的一个LFSR状态,在加载进nt=0x0000XXX0,当加载reader随机数nr的时候,LFSR的这些零

被左移,产生一个如0x000YZZZZZZZZ格式的LFSR状态.

are shifted to the left, producing an LFSR state of the form `0x000YZZZZZZZZ`.

一旦我们在我们的表格中有这些格式的所有LFSR状态.我们就能够恢复它.通过搜索ks2.ks3.

searching for $\mathsf{ks}_2, \mathsf{ks}_3$.

Typic 通常情况下,在我们的表中,0xXXX击中一个值.因为keystream的大小是64位,而LFSR只有48位.在section 6.2我们显示我们怎样能够使用LFSR状态,我们在表中发现的.nt和nr@ks1一起,得到安全密钥.

the size

Section

together

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 $r_k r_{k+1} \ldots r_{k+47}$ 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 $r_{k-1} r_k \ldots r_{k+46}$.

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 $b_0b_1 \dots b_{n-1}$ 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 $\bar{r} = r_0r_1 \dots r_{46+n}$ of bits such that

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

and such that

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

Condition (2) says that $\bar{r}$ is generated by the LFSR, i.e., that $r_0r_1 \dots r_{47}$, $r_1r_2 \dots r_{48}$, $\dots$ 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 $s_0s_1 \dots s_{19}$ 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,

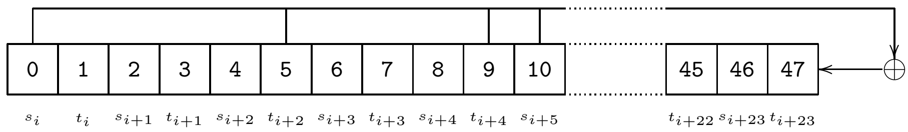


Fig. 9. Subsequences $\bar{s}$ and $\bar{t}$

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

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

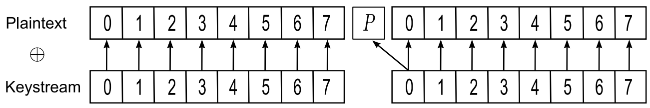


Fig. 10. Encryption of parity bits

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 $b_0^{1,\bar{s}} \ldots b_{n/2-5}^{1,\bar{s}} \tilde{b}_0^{1,\bar{s}} \ldots \tilde{b}_{n/2-5}^{1,\bar{s}}$, and the second table by $b_0^{2,\bar{t}} \ldots b_{n/2-5}^{2,\bar{t}} \tilde{b}_0^{2,\bar{t}} \ldots \tilde{b}_{n/2-5}^{2,\bar{t}}$.

Since $s_0 t_0 s_1 \ldots t_{n/2}$ is generated by the LFSR if and only if $b^{1,\bar{s}}\tilde{b}^{1,\bar{s}} = b^{2,\bar{t}}\tilde{b}^{2,\bar{t}}$ 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 $r_9 r_{10} \ldots r_{46+n}$. So to obtain the state of the LFSR at the start of the keystream, we have to roll back the state $r_9 r_{10} \ldots r_{58}$ 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)