

# Making the Best of Mifare Classic

Wouter Teepe  
w.teepe@cs.ru.nl

Radboud University Nijmegen  
October 6, 2008

## Abstract

What would you do if you would be instructed to make a secure application built on the Mifare Classic? Arguably, due to the vulnerabilities shown in [5], this is rather difficult and it may be easier to use to another chip. This document explores what the best is you can get, if the only option is Mifare Classic. We propose countermeasures against state restoration and against cloning. The effectiveness of these countermeasures depends on the absence of other vulnerabilities of the Mifare Classic.

## 1 Introduction

In [5], it is shown that the proprietary cryptography used on the Mifare Classic RFID chip is severely flawed. The management summary would be something like “Mifare Classic is broken”. And in fact, none of the authors would probably strongly recommend using the Mifare Classic while alternatives are available. The scientific summary would however be something like “Mifare Classic adheres to a different, weaker, security model”. Arguably, this is a euphemism, but technically it is correct. The adjusted security model provides hardly any security at all. But still, it is more than nothing at all.

There are two motivations for investigating the remaining security features. One is academic, and one is practical.

Academically, one would like to answer the question what the absolute minimum requirement of an RFID card is to be able to use it for anything requiring *some* level of security. It turns out, that the Mifare Classic, with an adjusted security model due to [5] can still offer protection against cloning and state restoration (reversing the state to one earlier recorded). Semantic security of the data stored must be achieved by using cryptography external to the Mifare Classic, such as AES.

Practically, there may be cases where for some reason or another, migration from Mifare Classic to another card is impossible, or infeasible on the short term. For those situations, it is good to have insight in how the Mifare Classic, given its vulnerabilities, can best be applied. This document provides suggestions for that. However, in some cases it may turn out that the resources one must spend to implement these suggestions might be better directed to replacing the Mifare Classic card after all. In other cases, implementing these countermeasures may buy sufficient time to prepare a card replacement in a later stage.

Moreover, it may be worthwhile to implement our countermeasures also on the future replacement card. This would pre-emptively add extra layers of security, which may turn out to be needed might the replacement card turn out to have weaknesses as well at some point in the future. Thus, implementation of our countermeasures where it is not strictly needed creates redundant security.

Many potential countermeasures can be identified. However, some countermeasures can be considered “teasing the attacker”, in the sense that attacks are not made impossible, but only slightly more cumbersome. This paper does not focus on such countermeasures. On the other hand, we focus only on countermeasures which fundamentally undercut the premises of successful attacks.

Our countermeasures go not without a strong disclaimer, however. Firstly, we might have overlooked things. Peer review and feedback will hopefully address that issue. If you find problems, please contact the author. Secondly, the Mifare Classic card may turn out to have more vulnerabilities than those published so far. In the light of additional vulnerabilities, the countermeasures may not work anymore. Thirdly, we provide the countermeasures on an “as is” basis and accept no liability for any damages resulting from using them.

The results from [5] show that keys can be retrieved from genuine card readers, and from intercepted communication between a card and a reader. For one, this leaves the data on the card open for everyone to read. Keys which have write rights leave the corresponding sectors of the card open to manipulation.

More fundamentally, one can identify two main classes of attacks resulting from these vulnerabilities.

**State restoration** This is an attack where the attacker obtains a card, and manipulates it to his advantage. The attacker reads the card state. Then he uses the card, typically depleting the monetary value stored on the card. When the card is depleted, the attacker writes back the original card state, and restored the monetary value he just spent.

**Cloning** This is an attack where the attacker selects a “victim card”, and makes functional copies of that card to his own advantage. The attacker reads the card state of the victim card, and loads this state into a Mifare Classic emulator. There is no limitation on how many clones the attacker can make, except for the number of available emulators.

For both attacks, we propose countermeasures.

State restoration can be prevented by tying the card state with a cryptographic signature to a monotonically decreasing counter on the card. This requires some data infrastructure, key infrastructure and allocation schemes on the card to make everything work. The Mifare Classic does not provide ACID write transactions, which complicates this considerably.

Cloning can be prevented by making the operation of a card dependent on reading data for which read keys are not yet disclosed to the attacker. Essentially this requires for every transaction an isolated part of the card memory. Here we distinguish two main application cases. In access control to buildings and facilities with online readers this can be used to prevent attempted clones from giving access to the premises. This requires the card to be reminted frequently in a secure environment, however. In the public transport ticketing application, with semi-offline readers, this allows the readers to distinguish the victim card

from the attempted clones. The victim card can be reminded on the fly in such a way that the attempted clones can be blacklisted. In this application case, a window of opportunity for the attacker remains. Our solution does not blacklist the card of legitimate rule-abiding innocent travellers. This is a major advantage over the straightforward blacklisting which is currently employed by transport ticketing systems such as the OV-chipkaart and the Oyster Card.

Fundamental to both countermeasures is that we trade storage space and transaction time for improved security. We do not propose one-size-fits all countermeasures, but explain in detail the many design choices one can make while implementing the countermeasures. What the best choice is depends on the particular properties of the application case at hand, and may be different from case to case.

Note that even with these countermeasures, relay attacks remain possible. This is not unique for the Mifare Classic: no currently available card is protected against relay attacks.

This paper is organised as follows. Section 2 describes the specifics of the Mifare Classic in sufficient detail to understand the countermeasures we propose. Section 3 describes our countermeasure against state restoration attacks. Section 4 discusses what it takes to create a functional clone, and our countermeasure against cloning. By that time, the reader will have read about a large number of keys and master keys required to make everything work. In Section 5, a summary is given of all the possible keys that may be involved in our countermeasures. We end with some concluding remarks and acknowledgements.

## 2 The Mifare Classic

### 2.1 Configuring

In this section it is explained what ways the Mifare Classic chip can be configured. The main source is the documentation provided by NXP [6]. However, the information has been reshuffled considerably and interpreted resulting in the exposé given in this section. In particular Figure 1 and Table 2 have no direct counterpart in the NXP documentation. In some cases, the documentation is ambiguous and lab experiments were conducted to complete the picture. These completions are marked and should be handled with care, as different hardware revisions might handle those cases differently.

The explanation given in this Section also holds for chips and devices which can emulate the Mifare Classic, such as the Mifare Plus<sup>1</sup> and the SmartMX from NXP, the “M”-labeled chips in the SLE66 series of Infineon, and the Proxmark3 programmed by Gerhard de Koning Gans. However, it may be that these emulating chips and devices offer alternative means to access the memory on the emulated chip. These alternative means are out of the scope of this section, but have to be within scope when one builds a security architecture using such an emulator.

In this section, only the publicly available documentation of the Mifare Classic is taken into account. There may be confidential information which changes the proper interpretation. In particular, if a Mifare Classic chip can be restored

---

<sup>1</sup>Projected by NXP in Q4 2008.

name	sectors		blocks		memory		slack space	
	64 b	256 b	total	free	total	free	min	max
mini	5	0	20	14	320 b	224 b	5 b	35 b
1K	16	0	64	47	1024 b	752 b	7 b	112 b
4K	32	8	256	215	4096 b	3440 b	40 b	280 b
Fudan	64	0	256	191	4096 b	3056 b	64 b	448 b

Table 1: Variants of the Mifare Classic

to factory settings using some undocumented feature, assessments that a chip can be “frozen” into particular states do no longer hold.

**On sectors and blocks** The Mifare Classic chip is essentially a memory card. One can store data on it, and later read out the data. The chip itself cannot execute user-loadable programs, for example in the way that Java cards can. However, the chip has some hardwired logic circuitry that can be configured to regulate the access to the card. This section will explain the functional possibilities of this circuitry and what configuration options exist.

There are several variants of the Mifare Classic chip, whose only essential difference is the storage capacity. The memory of every Mifare Classic chip is divided into a number of *sectors*, each of which can be configured independently. A sector is the biggest unit of a chip that can be configured; a sector operates independently from other sectors on the same chip. Therefore, in the rest of this section we consider the individual sector of a Mifare Classic chip as the subject of discourse.

Sectors on a single chip are numbered consecutively, starting at 0. Every sector is divided into blocks of 16 bytes. Not all blocks can be used for storage of arbitrary data. Of every sector, the last block, the *sector trailer*, is reserved for configuring the sector. Moreover, the first block of the first sector, the *manufacturer block*, of a chip is read-only and is initialised by the manufacturer. It contains the hardware version number and the (globally unique) serial number of the chip. The number and sizes of the sectors differ per variant of the chip. A summary is given in Table 1. The Fudan chip is an unlicensed clone (“counterfeit”) produced by the China-based company Fudan Microelectronics Co. Ltd.

The sector trailer is, as any block, 16 bytes long. The actual configuration is stored redundantly in 3 bytes. The actual configuration space is 12 bits long; these bits are called the *access bits*. Key A takes 6 bytes. Depending on the configuration, there are 7 free bytes, or 1 free byte and a key B which takes another 6 bytes. The way to change the configuration and the key(s) is simply to write the desired configuration to the sector trailer block.

The granularity of access to the memory in a sector is the block. A “read” command returns the contents of a whole block; a “write” command overwrites a whole block. Thus, if one would like to change only one single byte, one has to read the whole block, change the single byte, and write back the whole block to the chip. For the sector trailer, there is an extra complication: a read of the sector trailer returns zero on the byte positions where the keys are stored. Thus, to change the configuration *without* changing the keys, one has to know the keys in order to construct the 16 bytes that must be written to

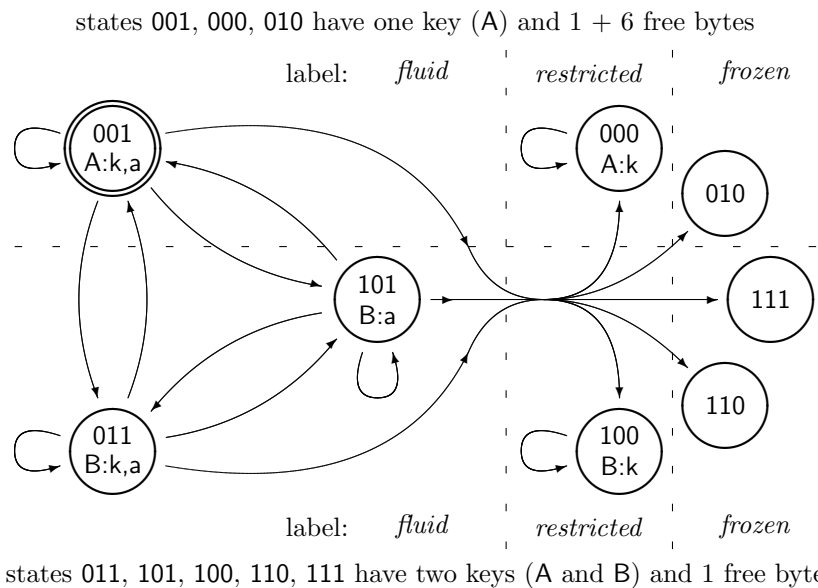


Figure 1: State transition diagram of the trailer access bits  $C_{13}C_{23}C_{33}$ .

the sector trailer. Moreover, one cannot recover an unknown key B by changing the configuration and then reading the corresponding bytes, since changing the configuration required one to overwrite those very same bytes.

A sector is divided into four distinct parts, for which the access conditions can be set independently of one another. In case of a 64 byte sector, every part consists of one block. In case of a 256 byte sector, there are three parts of 5 sectors each, and one part which consists of the sector trailer only. A sector has 12 *access bits*, which are four groups of three bits each; one group for every part of the sector. The first three groups ( $C_{10}C_{20}C_{30}$ ,  $C_{11}C_{21}C_{31}$ ,  $C_{12}C_{22}C_{32}$ ) contains *data access bits*, the last group ( $C_{13}C_{23}C_{33}$ ) contains the *trailer access bits*.

**On trailer access bits** The three bits  $C_{13}C_{23}C_{33}$  which control the sector trailer are particularly interesting, as they configure the configurability of the sector: as long as one can write to the access bits, one can change the configuration of the sector. The 8 possible states of the trailer access bits are depicted in the state transition diagram<sup>2</sup> in Figure 1. Every circle denotes a state. State 001 is the state in which the sector is when it leaves the factory. When it is possible to change the sector trailer while in a particular state, the particular state is annotated with key (A or B) that is required to change the state. The key is annotated with the kinds of changes that can be made from that particular state (i.e. its rights). Annotation with the right k means, depending on the context, that it is possible to change/overwrite keys A and B, or key A and the 6 free bytes; annotation with the right a means that it is possible to change the 12 access bits and the one free byte.

<sup>2</sup>Figure 1 has been created by manual interpretation of Table 3 of the NXP documentation of the Mifare Classic 4K [6].

$C_1C_2C_3$	one-key, i.e. 000, 001*, 010		two-key, i.e. 011, 100, 101, 110, 111			
	$C_1C_2C_3$	A	label	A	B	label
000*	r w d i		<i>fluid</i>	r w d i	r w d i	<i>fluid</i>
110	r d		<i>restricted</i>	r d	r w d i	<i>fluid</i>
100	r		<i>frozen</i>	r	r w	<i>fluid</i>
011			<i>dead</i>		r w	<i>fluid</i>
001	r d		<i>restricted</i>	r d	r d	<i>restricted</i>
010	r		<i>frozen</i>	r	r	<i>frozen</i>
101			<i>dead</i>		r	<i>frozen</i>
111			<i>dead</i>			<i>dead</i>

Table 2: Data access bits. The factory defaults are marked with \*.

For example, from state 001 one can change both the keys and the access bits using key A; from state 000 one can change only the keys, but nothing else. State 101 is particular that one can change the configuration, but not the keys. However, in two steps from state 101 one *can* also change the keys.

Also depicted in Figure 1 is the number of keys that a configuration has: the upper three states have only one key (A), the lower three also have a second key (B) at the expense of 6 free bytes. It can be seen that there the states 110 and 111 are equivalent. Not depicted in the figure are the read rights. Any valid key gives read permission to the access bits and the free bytes.

As long as the trailer access bits are in state 001, 011 or 101, the trailer is *fluid*, it can be changed in any other possible state, given the correct key. In states 000 and 100, trailer is *restricted*, the access bits and the free bytes can no longer be changed, but the keys can still be changed. In states 010, 110 and 111, the trailer *frozen*, it is in an unmodifiable state.

Note that it is not possible to give a key write access to the free bytes without giving the same key total control over the complete sector.

Experiments show that in configurations which have one key and 6 free bytes, one can guess the value of the 6 free bytes and get a yes/no answer from the card. This is done by an authentication attempt for the non-existing key B, where the guess is used as key. If the authentication attempt is successful, the guess equals the 6 free bytes; otherwise, it is not equal. After successful authentication, there appear to be no valid commands. This means that in those configurations one can access the 6 bytes without knowledge of key A, which clearly limits the secrecy guarantees of those bytes.<sup>3</sup>

Thus, depending on the configuration, there are per sector 1 or 7 bytes available in the trailer, but their use has some serious limitations. Therefore, in Table 1, we refer to these bytes as *slack space*.

**On data access bits** There are three sets ( $j \in \{0, 1, 2\}$ ) of three bits each  $C_1C_2C_3$  which control the configurations of the data blocks. Depending on the sector size, a group of three bits controls either the rights to one single block, or to 5 blocks of the sector. Essentially, a configuration is a mapping of rights to keys. Once authenticated against a key, there are four possible logical

<sup>3</sup>Only after extensive exegesis, this behaviour can be inferred from the NXP documentation [6].

operations, which coincide with the four distinctive rights: *read* (*r*), *write* (*w*), *decrement* (*d*) and *increment* (*i*).<sup>4</sup>

There are 8 distinctive modes for  $C_1C_2C_3$  which distribute rights to the keys A and B. However,  $C_1C_2C_3$  determines whether key B actually exists. Therefore, one can distinguish 16 distinct configuration modes, which are given in Table 2. This table gives for every combination of  $C_1C_2C_3$  and  $C_1C_2C_3$  the logical rights of all existing keys in that particular configuration. It can be seen that in the one-key configuration, there are only four distinct right configurations. In total, a block can thus be configured in 12 distinct ways.

All 16 modes have been labelled: *dead* means that the block cannot be accessed at all; *frozen* means that the block is read-only; *restricted* means that the block contents can be changed, but not to all possible states; *fluid* means that the block contents can be changed to any other state.

All modes that carry the label *restricted*, have for  $C_1C_2C_3$  either the value 110 or 001. The NXP documentation [6] advertises these modes as “value blocks”, which are suitable for “electronic purse applications”. There are restrictions on the memory modifications possible in these modes, which will be discussed in the next section.

**Impossibilities** Obviously, there are many things the Mifare Classic chip cannot do. However, some of these limitations may not be very obvious. We will mention a few.

In a two-key configuration, the key B is always strictly more powerful than key A. Thus, one cannot create a sector in which (for example) only key A can access block 1, and only key B can access block 2. Therefore, one cannot use smart tricks to divide a sector up into the equivalent of two smaller single-key sectors.

There is no configuration which supports ‘blind writing’, i.e. that a key has write access to a block, but no read access. The only thing that can be written without being readable are the keys themselves; however, these written keys cannot be recovered by the use of another more powerful key.

## 2.2 Using Value Blocks

A block that is in the “value block” mode stores a 32-bit signed integer. Instead of read and write commands, the block offers increment and decrement commands. Key A can only perform decrement operations. Key B, if it exists, can perform any operation if the block is in mode 110. This is at least the general working of a value block. Obviously, its contents can be changed, but not into any other state desired. Therefore we consider value blocks to be *restricted*. In practice, the modification possible on a value block are subject to some subtle conditions. In this section we will elaborate on these conditions.

---

<sup>4</sup>Note that the actual command set contains also the commands *transfer* and *restore*. Technically, the *decrement*, *increment* and *restore* do not change the non-volatile memory of the chip, a subsequent *transfer* command is needed to write the result of these commands to non-volatile memory. As the right to perform the *transfer* command is implied by the right to perform *decrement*, *increment* or *restore*, the *transfer* right can be omitted from our analysis. The *restore* command simply prepares the current value of a block for being re-written to non-volatile memory. From the perspective of logical memory access, it is equivalent to a null operation, and therefore omitted from our analysis.

Modification of value blocks always goes via a temporary register on the card. There are three documented commands that write data from a value block into the temporary register: increment, decrement and restore. The increment and decrement modify the value accordingly before storing it in the register. The restore command does not modify the value. There is one command which writes from the temporary register to a value block: transfer.

All four commands take a parameter which specifies to which block the operation should apply. It is thus possible to increment on block  $x$ , then transfer on block  $y$ ; which will leave block  $x$  unchanged but which will overwrite block  $y$ .<sup>5</sup>

The actual block contains the 32-bit signed integer stored redundantly, and an extra byte also stored redundantly (the latter is called “ADR” in the NXP documentation [6]). The four commands all transport the ADR byte without changing it. Thus, if a value is transported from one block to another block using value block commands, the ADR byte is transported as well.<sup>6</sup> The only way to change an ADR byte except for taking it from another value block, is to do a write operation on the block. Of course, a write operation is not permitted on a value block, but it is permitted to write to a block in a *fluid* state, and then change the mode to a value block after that.

As said, there are a number of restrictions on how a value block can be modified.

1. Firstly, the *increment*, *decrement* and *restore* commands are only executed in a consistent state (i.e. the redundant copies of the value do match), and always result in a consistent state.<sup>7</sup>
2. Secondly, the *increment* and *decrement* do not allow arbitrary operands. Both commands take a 4-byte signed integer, but ignore the sign bit of the operand; effectively only positive operands are allowed.<sup>8</sup> This is intuitive in the sense that an increment command cannot effectively decrement a value, nor vice versa.
3. Thirdly, the *increment* and *decrement* commands refuse to overflow the value over  $\text{maxint} (2^{(32-1)} - 1)$  or underflow the value below  $\text{minint} (2^{(32-1)})$ .<sup>9</sup> This is intuitive in the sense that an incrementing a positive value cannot result in a negative value, nor vice versa.
4. Fourth, the commands are only performed if they are actually allowed by the configuration.

---

<sup>5</sup>The possibility to move values around like this is not discussed in the NXP documentation [6], but observed from experiments, and confirmed as intentional by NXP in personal communication.

<sup>6</sup>This transportation of the ADR byte is not mentioned in the NXP documentation, but observed from experiments.

<sup>7</sup>Experiments show that if the card is in an inconsistent state and an increment, decrement or restore is performed, the card returns an error, halts the card and does not change the non-volatile memory. The NXP documentation however is ambiguous to what happens in such a case.

<sup>8</sup>This follows from experiments with the chip. The NXP documentation is ambiguous to what happens if these commands are given a negative operand.

<sup>9</sup>This follows from experiments with the chip. The NXP documentation [6] is ambiguous as to what happens in such a case. Moreover, the NXP documentation is not explicit on the values of  $\text{maxint}$  and  $\text{minint}$ , though these values are logical choices.

5. Fifth, after an increment, decrement or restore, transfer is the only allowed command, and transfer is forbidden in any other state. At least, the NXP documentation offers a command transition diagram which suggest this. This implies that it is impossible to move the contents of value blocks over the boundary of the individual sector: after authenticating for another sector, first an increment, decrement or restore is required before a transfer can be performed. This prevents the temporary register to serve as a means to move data from one sector to the other.

**Making a value block really restricted** It can be said that though the value block does not allow arbitrary write commands to it, special care has to be taken in order to make sure that it cannot be modified arbitrarily by indirect means. We identify two indirect means.

The first option is to copy a value from another value block in the same sector (i.e. one in mode 110 or 001), using a *restore* and a *transfer*. This cannot be prevented, but it can be detected by making sure all value blocks in a sector have a different ADR byte.

The second option is to copy a value from another block in the same sector which is in state 000, also using the *restore/transfer* combo. If such a block exists, this cannot be prevented, and it cannot be detected, as the adversary can write any information he desires to the originating block and then copy it, including the ADR byte. This attack can be prevented by making sure there simply are no blocks in the sector in state 000: then there is nothing to copy *from*.

Of course, it has to be made sure that the state of the blocks cannot be modified by the adversary by setting the appropriate trailer access bits.

**An undocumented feature** Karsten Nohl has been so kind as to inform me that the fifth restriction mentioned above does not hold. The undocumented feature is that the command transition diagram given in the public documentation of NXP [6] is not complete. In some hardware versions, after a *decrement* command, one can perform a *read*. This can read the contents of another block in the same sector into the temporary buffer. After such a *read*, one can perform a *transfer* which writes the temporary buffer back to the block on which the original *decrement* command was given. In this manner, a value block which is to be in a *restricted* mode can be overwritten with any value that can be constructed in a readable block in the same sector.

This feature can be circumvented. Instead of avoiding just particular other blocks within one sector, just avoid using *any other* block in the same sector. That is, configure any desired number of value blocks reflecting the same value within one sector, and configure all other blocks in the sector as *dead* (e.g. mode 111).

The adjusted command transition diagram is given in Figure 2. We have no reason to believe that this adjusted transition diagram is complete. The diagram should therefore be treated with caution. This undocumented feature should be taken as a stiff warning that there may be other showstopping features waiting to be discovered.

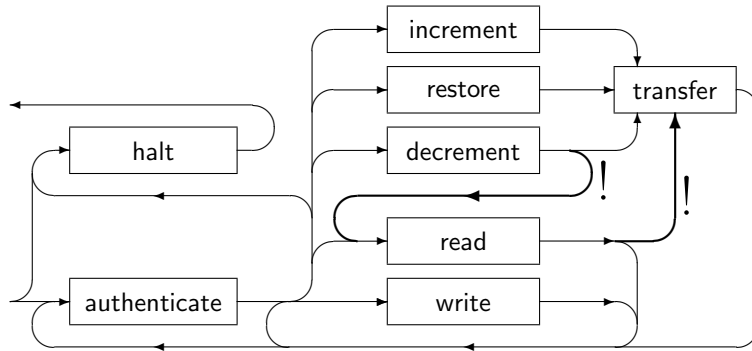


Figure 2: Command transition diagram of the Mifare Classic. The two transitions in this diagram with an exclamation mark have experimentally been observed to exist, but are not previously (publicly) documented.

### 3 Preventing Restoration of Previous States

There are various attack scenarios to the Mifare Classic, depending on the application context of the card. In some applications a card is stateful, and restoration to a previous state may enable a criminal business case, or may at least enable behaviour that should not be possible. Examples of such stateful applications are ticketing systems such as the OV-chipkaart and the Oyster Card, where the card holds a monetary value. An exploit using state restoration would have this life-cycle: A card which is in the desired state (“fully charged”) is obtained and its state is read out completely. Then, the card is used until the card is in an undesirable state (“empty”). Then the original state is restored, used again, restored, used again, and so forth.

We propose a method for preventing the restoration of previous states. The idea is roughly the following.

**key infrastructure** All genuine state-modifying readers are supposed to be trusted, and there is a public-key infrastructure such that every genuine state-modifying reader can sign data, and that every genuine reader can verify signatures made by other genuine readers.

**card state signing** As a part of every transaction which modifies the state of a card, a modification is made on the card which is irreversible. Moreover, a cryptographic signature over the card status, including the status of the irreversible part of the card, is written onto the card by the reader.

**card state verification** When a reader encounters a card, it will verify the signature before it engages into a transaction with the card.

The irreversible modification on a card can be accommodated by using the access bits of one sector in a particular configuration. To be able to detect illegally modified cards, the changes to the irreversible block have to be tied to the rest of the state of the chip. The signature over the whole card status establishes this tie.

Note that the cryptography used in this solution is external to the Mifare Classic chip: the chip only stores data which happens to exhibit cryptographic

name	C <sub>1</sub> C <sub>2</sub> C <sub>3</sub>	C <sub>1</sub> C <sub>2</sub> C <sub>3</sub>	A	B
I	001	100	r d	r d k
II	001	110 or 111	r d	r d
III	001 or 110	000	r d k	
IV	001 or 110	010	r d	
V	110	100	r d	r d w i k
VI	110	110 or 111	r d	r d w i

Table 3: Configurations in which data block group  $j$  is *restricted* and the trailer is either *restricted* or *frozen*, all with respect to key A. For configurations I to IV, this is also with respect to key B. The rights  $r$ ,  $d$ ,  $w$  and  $i$  are with respect to block group  $j$ , the right  $k$  is with respect to the trailer.

properties enforced and checked by the readers. The only guarantee that the chip has to provide is the irreversibility of one particular block.

Let us assume the case where an adversary is able to retrieve all relevant cryptographic keys giving access to the Mifare Classic chip, the adversary knows all previous states of the card, but the adversary has no access to a private key of a reader. The adversary can change the status of the card. There are many possible card states which have a valid signature. There is only one accessible state of the card in which the card has a valid signature, which is the current state. Thus, this adversary cannot change the card in any manner that a genuine reader would accept.

Using a public-key infrastructure, there are two classes of genuine readers:

**verification readers** These readers have only public verification keys, and thus can verify the validity of a card. However, these readers have no private keys, and thus cannot perform valid state-modifying transactions.

**state-modifying readers** These readers have public verification keys, and private signing keys. These readers can verify the validity of a card, and can modify cards into new valid states.

Typically, compromised verification readers pose no security risk, as they only contain public information. Therefore, verification readers can be distributed abundantly if so is desired; everybody can verify whether a card is valid. In the case that there is no need for a distinctive class of verification readers, the key infrastructure can be simplified as follows: All state-modifying readers have a shared symmetric key. The ‘signature’ is then the cryptographic hash of the card status, encrypted under the shared key, or simply a MAC.

When one chooses for such a symmetric setup, one trades computation power of the reader and storage space on the card (more on that later) for the possibility to have readers which can verify the card state without changing it. Here, we will continue our explanation as if an asymmetric setup is chosen, knowing it can be trivially changed into a symmetric setup.

Verification of the integrity of the state of a card is of course also a crude but effective way of performing input validation on the application level data on the card.

**Practical implementation** There are a number of practical hurdles to take for implementing the proposed solution. We will address and resolve them one

by one.

Firstly, an ‘irreversible’ mode of the card is needed. This can be done by setting the access bits of one sector in such a configuration that (1) the access bits cannot be changed thereafter (the trailer is *restricted* or *frozen*), and (2) one block of the sector can only be changed ‘in one direction’ (the block is *restricted*). Configurations I through IV given in Table 3 list all such states possible with the Mifare Classic chip. In configurations V and VI given in Table 3, the data block is *restricted* with respect to key A, but *fluid* with respect to key B. In these configurations, the irreversible block is a ‘value block’ which contains a 32-bit signed integer. which can only be decremented.

The number of transactions possible is however not  $2^{32} - 1$ , but only about  $10^5$ , the write endurance of the Mifare Classic chip (the number of times the EEPROM can be overwritten before being worn out). The Mifare Classic chip has a data retention time of 10 years, which is probably longer than the lifespan of a single deployed card. Assuming a deployment time of 10 years, this would allow an average of 27 transactions per day, every day, for 10 consecutive years, including weekends and holidays. This will be sufficient for most applications that we can think of at the moment.

Secondly, a transaction mechanism is needed to switch from one valid state to the next valid state. The Mifare Classic chip is in this respect a rather simple chip which does not have special features such as ACID transactions spanning changes to multiple blocks on the chip. The Mifare Classic does not even support something like an ACID transaction on a single block: When a card is taken out of the radiographic field at the wrong moment (tearing), a block can be left in an inconsistent state, one in which the block contents is neither the original state nor the projected to-be-written state. The projected Mifare Plus will support ACID transactions, but only on sector trailers and not on data or value blocks.<sup>10</sup>

The lack of atomicity in Mifare Classic write operations complicates our solution considerably. Nevertheless it is possible to implement our solution on Mifare Classic by using technologies which are similar to atomic commits in databases and journaled filesystems.

In our solution we distinguish between the *physical* state and the *logical* state of a card. The physical state is simply the complete memory contents. The logical state is a restricted view on this memory contents, based on a storage convention which is assumed to be used. The very same is the case with hard disks in computers, which have a physical state (a very long list of bits) and a logical view (a filesystem). The similarity is not a coincidence, but intentional. Our solution is essentially storage convention, or if you like: a primitive file system with atomic commits.

Central to our solution are the *countersector* which facilitates an irreversible state counter and acid transactions simultaneously, the *block allocation table* which relates physical card state to logical card state, and a *signature suite* which is used to mark valid states and to verify the integrity of states. The basic state transition mechanism looks like this:

1. **State verification** The logical card state is read out and verified. If the card is not *valid* but in a *recoverable* state, the state is recovered. If the

---

<sup>10</sup>Personal communication with NXP, June 11 2008.

reader cannot recover the card, the transaction is aborted. (More on these states in the next section.)

2. **Valid state** The card is in a consistent state. The countersector is valid and the signature verifies. Some data blocks are unused.
3. **Transaction preparation** Into the unused blocks, new information is written such that would the state counter be decremented, a next valid state would be reached. This new information includes a new signature over the new card state.
4. **Transaction commit** The state counter is decremented, atomically.
5. **Next valid state** The card is in a new consistent state. Some (other) blocks are unused.

In the following sections, we elaborate on the details of the building blocks of this solution.

### 3.1 Countersector

The countersector employs a mechanism to make the card resistant against tearing. This is achieved by having multiple copies of the counter, dispersed over multiple blocks within one single sector. In normal use, all copies are decremented one by one. When tearing corrupts the countersector, recovery has to be done. There are essentially three different strategies for this, all of which will be presented.

The first strategy (RU1) has been developed by us (Radboud University). We discussed our methods with NXP, who discussed them with TNO. TNO came up with a variation (TNO-RU), which is also presented. Moreover, this led to a variation of our own strategy as well (RU2). Each strategy has its own pros and cons, which will be discussed later on.

The countersector consists of three *value blocks* which are in configuration V or VI of Table 3. Thus, these blocks store a 32-bit integer which key A can only decrement. Key B can also increment or overwrite these blocks, but key B is not used nor present in normal readers. We number these blocks 1, 2 and 3.

The countersector can be in many states. We distinguish three classes of states. *Valid* states are the states in which blocks 1, 2 and 3 of the countersector hold one and the same number. *Recoverable* states are the states from which a valid state can unambiguously be derived, these are given in Table 4. *Corrupted* states are all the other states. As long as only the (pseudo)code of this paragraph accesses the countersector, the countersector will never enter a corrupted state.

Between RU, NXP and TNO there have been some discussions as to how many copies of the counter are needed. The original RU strategy has three copies. In this manner, it is possible to reconstruct which state should be the next state, essentially by doing a kind of majority vote on the counters. TNO pointed out that one can also do with only two copies, and use the card data to test which of the two counters matches the corresponding cryptographic signature elsewhere on the card. This requires substantially more read operations when a recovery is done (which is hopefully rarely), but saves one decrement operation for every normal transaction (which is obviously often). It does not

state	valid	contents of block			recovery strategy			
		1	2	3	RU1		RU2	TNO-RU
					A	B	A	A
0	✓	$s$	$s$	$s$				
1		other	$s$	$s$		c u		c u
2		$s - 1$	$s$	$s$	c	c u	c	c u
3		$s - 1$	other	$s$		c u		c u
4		$s - 1$	$s - 1$	$s$	c	c u	c	c u
5		$s - 1$	$s - 1$	other		c u		c
6	✓	$s - 1$	$s - 1$	$s - 1$				

Table 4: Possible states of a countersector after a transaction which may have been interrupted. The state counter is denoted with  $s$ . *Other* means “neither  $s$  nor  $s - 1$ ”. There are three possible recovery strategies. For each of them, it is given which key can recover in which direction from which state. The capabilities c stands for commit and u stands for undo/rollback.

```

function go-to-next-state(first-block, old-state) → success
  for  $j \leftarrow$  first-block up to 3 do
    repeat
      decrement( $j$ , 1);
      transfer( $j$ );
       $n \leftarrow$  read-block( $j$ );
    until  $n \neq$  old-state;
    if  $n \neq$  old-state - 1 then return false;
  return true;

```

```

function commit-transaction() → success
  old-state ← read-block(1);
  return go-to-next-state(1, old-state);

```

Figure 3: Pseudocode of the commit transaction with key A.

```

function RU1-RU2-recover-commit-with-key-a() → success
   $b1 \leftarrow$  read-block(1);  $b2 \leftarrow$  read-block(2);  $b3 \leftarrow$  read-block(3);
  if  $b1 = b2 = b3$  then return true; /* state 0 or 6 */
  if  $b2 = b3$  then
    if  $b1 \neq b2 - 1$  then return false; /* state 1 */
    return go-to-next-state(2,  $b2$ ); /* state 2 */
  if  $b1 = b3 - 1$  then
    if  $b1 \neq b2$  then return false; /* state 3 */
    return go-to-next-state(3,  $b2$ ); /* state 4 */
  return false; /* state 5 or not in Table 4 */

```

Figure 4: RU1, RU2: Pseudocode of the commit recovery with key A. States 0, 2, 4 and 6 of Table 4 are recognized and recovered if needed. Otherwise, the function returns false.

```

function RU1-recover-commit-with-key-b()  $\rightarrow$  success
   $b1 \leftarrow$  read-block(1);  $b2 \leftarrow$  read-block(2);  $b3 \leftarrow$  read-block(3);
  if  $b1 = b2 = b3$  then return true; /* state 0 or 6 */
  if  $b2 = b3$  then
     $new-state \leftarrow b2 - 1$ ; /* state 1 or 2 */
  else
     $new-state \leftarrow b1$ ; /* state 3 or 4 or 5 */
    if  $(b1 \neq b3 - 1)$  and  $(b1 \neq b2)$  then
      return false; /* state not in Table 4 */
     $first-block \leftarrow 1$ ; /* state 1 */
  if  $b1 = new-state$  then  $first-block \leftarrow 2$ ; /* state 2 or 3 */
  if  $b2 = new-state$  then  $first-block \leftarrow 3$ ; /* state 4 or 5 */
  for  $j \leftarrow first-block$  up to 3 do
    repeat
      write-block( $j$ ,  $new-state$ );
       $n \leftarrow$  read-block( $j$ );
    until  $n = new-state$ ;
  return true;

```

Figure 5: RU1: Pseudocode of the commit recovery with key B. All states of Table 4 are recognized and recovered. Otherwise, the function returns false.

```

function RU1-recover-rollback-with-key-b()  $\rightarrow$  success
   $b1 \leftarrow$  read-block(1);  $b2 \leftarrow$  read-block(2);  $b3 \leftarrow$  read-block(3);
  if  $b1 = b2 = b3$  then return true; /* state 0 or 6 */
   $new-state \leftarrow b3$ ; /* state 1 or 2 or 3 or 4 */
  if  $b2 = b3$  then
     $last-block \leftarrow 1$ ; /* state 1 or 2 */
  else
    if  $b1 = b3 - 1$  then
       $last-block \leftarrow 2$ ; /* state 3 or 4 */
    else
      if  $b1 = b2$  then
         $new-state \leftarrow b1 + 1$ ; /* state 5 */
         $last-block \leftarrow 3$ ;
      else
        return false; /* state not in Table 4 */
  for  $j \leftarrow last-block$  down to 1 do
    repeat
      write-block( $j$ ,  $new-state$ );
       $n \leftarrow$  read-block( $j$ );
    until  $n = new-state$ ;
  return true;

```

Figure 6: RU1: Pseudocode of the rollback recovery with key B. All states of Table 4 are recognized and recovered. Otherwise, the function returns false.

save storage space, as the third block in the countersector which could be saved may not be used for any other purpose, due to the special care that has to be taken when using value blocks. For the sake of simplicity and elegance, we present all three recovery strategies using three copies of the counter, knowing that the strategy can easily be changed into one using only two copies.

In a commit transaction, the blocks are decremented and read, one by one. When an error occurs, the procedure is simply aborted. The pseudocode for this procedure is given in Figure 3. If no errors occur, the countersector is again in a valid state.

Now what happens when the operation is interrupted somewhere, or when a decrement operation does not produce the desired result? If a decrement operation does not change the state at all, it is simply retried. If a decrement operation yields the wrong result, for whatever reason, the transaction is aborted by the reader. As a result, when a countersector is in a valid state undergoes the transaction, after the transaction the countersector will be in one of the states of Table 4.

Every possible resulting state can be recognized. States 0 and 6 are valid without further ado. From every invalid state it is possible to recover into a valid state. Recovery into a valid state can mean either an undo (rollback) to the state before the commit transaction, or a (re-)commit to the state intended by the original commit transaction.

There are three recovery strategies.

**RU1** In strategy RU1, changing the contents of a block is done by operations local to only that single block. Notably, this allows the ADR bytes of the blocks to be distinct. For states 2 and 4, recovery is done by using decrement transactions using key A, with pseudocode given in Figure 4. Key B can recover by using write operations from any state in Table 4, and in either direction, with pseudocode given in Figures 5 (commit) and 6 (undo).

**RU2** Strategy RU1 is essentially a mutilated version of strategy RU1. In RU2 there is no key B, and recovery from states 1, 3 and 5 is not possible. It is better than RU1 in the sense that there is no need for a key which has write rights. Recovery from states 2 and 4 is done with key A, with pseudocode given in Figure 4.

**TNO-RU** In strategy TNO-RU, changing the contents of a block is done by copying it from another block in the countersector, using the restore-transfer command combo. Notably, this strategy does not guarantee that the different block retain distinct ADR bytes. For this strategy only one key with the decrement right is needed. Recovery can be done both forward (commit) and backwards (undo/rollback), unless the last copy of the value has to be restored. In that case, only commit is possible. Pseudocode for commit and undo/rollback is given in Figures 7 and 8, respectively.

The pseudocode is already nontrivial as it is given. In reality, a number of checks and adjustments has to be built in to accommodate the fact that a value block is in fact not just a 32 bit integer, but a 16 byte array containing three redundant copies of the counter, and four redundant copies of the ADR byte.

```

function TNO-RU-recover-commit-with-key-a()  $\rightarrow$  success
   $b1 \leftarrow$  read-block(1);  $b2 \leftarrow$  read-block(2);  $b3 \leftarrow$  read-block(3);
  if  $b1 = b2 = b3$  then return true; /* state 0 or 6 */
  if  $b2 = b3$  then
    repeat /* state 1 or 2 */
      decrement(2, 1); /* register:  $b2-1$  */
      transfer(1); /* store in counter 1 */
       $n \leftarrow$  read-block(1);
    until  $n = b2 - 1$ ;
    proceed-from  $\leftarrow$  2;
    old-state  $\leftarrow$   $b2$ ;
  else
    if  $(b1 \neq b3 - 1)$  and  $(b1 \neq b2)$  then /* state not in Table 4 */
      return false;
    if  $b1 \neq b2$  then /* state 3 */
      transfer-to  $\leftarrow$  2;
      proceed-from  $\leftarrow$  3;
      old-state  $\leftarrow$   $b1$ ;
    else /* state 4 or 5 */
      transfer-to  $\leftarrow$  3;
      proceed-from  $\leftarrow$  4;
    repeat
      restore(1); /* register:  $b1$  */
      transfer(transfer-to); /* store in counter */
       $n \leftarrow$  read-block(transfer-to);
    until  $n = b1$ ;
  if proceed-from = 4 then /* state 4 or 5 */
    return true;
  else
    return go-to-next-state(proceed-from, old-state);

```

Figure 7: TNO-RU: Pseudocode of the commit recovery with key A. All states of Table 4 are recognized and recovered. Otherwise, the function returns false.

First of all, one has to redefine (overload) the equality “=” operator such that it returns false if either the redundant copies of the counter in one operand do not match, or the redundant copies of the ADR byte in one operand do not match. Even if the 16 byte operands are bit-wise equal. Moreover, the “=” operator ignores whether the ADR bytes of the two operands are mutually equal, but only compares the counter. The “ $\neq$ ” operation is to be read as an abbreviation of “**not** (... = ...)”, and thus inherits the overloading of “=”.

The arithmetic operator “-” applies to all redundant copies of the counter simultaneously, but not to the ADR byte.

In the RU1 and RU2 strategies, the ADR byte requires some special treatment. At initialisation, the ADR byte must be set to the block number it is stored in. The state is considered not valid if a block does not have the correct ADR byte. Whenever a block is being overwritten by either a transfer or a write command, the overwriting is only considered successful if the ADR byte

```

function TNO-RU-recover-rollback-with-key-b()  $\rightarrow$  success
   $b1 \leftarrow$  read-block(1);  $b2 \leftarrow$  read-block(2);  $b3 \leftarrow$  read-block(3);
  if  $b1 = b2 = b3$  then return true; /* state 0 or 6 */
  if  $b2 = b3$  then
     $last-block \leftarrow$  1; /* state 1 or 2 */
  else
    if  $b1 = b3 - 1$  then
       $last-block \leftarrow$  2; /* state 3 or 4 */
    else
      return false; /* state 5 or not in Table 4 */
  for  $j \leftarrow last-block$  down to 1 do
    repeat
      restore(3); /* register:  $b3$  */
      transfer( $j$ ); /* store in counter  $j$  */
       $n \leftarrow$  read-block( $j$ );
    until  $n = b3$ ;
  return true;

```

Figure 8: TNO-RU: Pseudocode of the rollback recovery with key A. states 1 to 4 of Table 4 are recognized and recovered. Otherwise, the function returns false.

contains the block number after the overwriting. In the recovery, if a block does not have the correct ADR byte, it is defined to be in the “other” state of Table 4, and thus can only be recovered using key B. In the recovery with key B, the ADR byte of the block that has to be overwritten must also be its block number. Of course, the actions with B cannot be performed in strategy RU2, as it does not cater a key B.

The recovery code of every strategy is constructed in such a way that if also the recovery is interrupted, the countersector is remains in a recoverable state. The transaction commit and recovery code jointly make sure that the countersector is always in either a valid state or a recoverable state. However, it might happen that the countersector is modified by other procedures into a state which is not in Table 4, for example by an adversary which has key A. In that case, the countersector cannot be recovered into a state which has a valid signature. Thus, an adversary may cripple the transaction system and with it the whole card, but he cannot put it to its own use.

**comparing the strategies** Essentially, the three strategies accomplish the same functionality. RU2 is functionally somewhat weaker than RU1 and TNO-RU because it cannot recover from all states in Table 4. Which criterions can be used to select one strategy over the other? This depends on the kind of redundancy and resistance one desires to build in. Let us not forget that the whole exercise of this paper is one of adding extra security layers, such that if some layers fail (as is the case with Mifare Classic), other layers remain. These added layers either guarantee the same level of security, or at least mitigate the risks that are the result of the failing layers. In fact, most if not all layers presented in this paper should be considered as mitigating, and not as providing

potential attack	recovery strategy		
	RU1	RU2	TNO-RU
card-only key retrieval		✓	✓
tearing block corruption	✓	✓	

Table 5: Resistance of countersector recovery strategies against potential new attacks on the Mifare Classic.

strong guarantees.

The lack of strong guarantees is inherent to the fact that the Mifare Classic may have yet undiscovered vulnerabilities. In the spirit of adding redundant security layers, it can be wise to speculate about possible undiscovered vulnerabilities. Here we will present two such vulnerabilities. It will turn out that the three strategies for recovery of the countersector differ considerably when one takes the resistance to these hypothetical vulnerabilities into account.

**card-only key retrieval** So far, all cryptographic attacks on Mifare Classic require a genuine reader which possesses the sector keys. From interacting with a reader, or intercepting the communication between a reader and a card, the keys can be recovered. So far, the card itself has stood cryptographic attacks rather well, as it requires the reader to authenticate first. However, it might be possible that given a card, one can choose a sector and then retrieve the sector keys, without destructing the card.

**tearing block corruption** It is known that if a card is teared out of the field while data is being written to a block, the contents of the block are undefined. Though the written data may is undefined, it could still be predictable, for example as a function of the timing of tearing (After how many nanoseconds the card is taken out of the field, or the field is switched off). Now assume that an adversary has a key which allows him to decrement a value block. Then, the adversary might try to switch of the field power at such a moment that the data being written is actually a valid value block, but only with a higher value. Then, the adversary has essentially incremented a block without possessing the increment right.

Now how could these potential attacks be used to increment the counters in the counter block?

In strategy RU1 key B has write permission. Using a card-only key retrieval, this key can be retrieved and the counters can be overwritten into any desired state. On the other hand, tearing block corruption in combination with an intercepted key A is less feasible. That is, the adversary has only one possibility to try this attack. If it fails, the block is in an inconsistent state and cannot be altered using only key A. One might try to perform the tearing block corruption attack by doing a restore on an uncorrupted block, and transfer to another “victim” block, but it is likely that this will modify the ADR byte, as all ADR bytes in the countersector are initialised distinctly. Moreover, once the attack is performed successfully, the incremented value block cannot be restore-transferred to the remaining countersector blocks, as this will inherently change the ADR bytes. Thus, strategy RU1 does not work in the face of a card-only key retrieval attack, but stands a decent chance against a tearing block corruption attack, thanks to the special treatment of the ADR byte.

In strategy TNO-RU there is no key with write permission, therefore a card-only key retrieval attack cannot produce a key with write permission, leaving the strategy resistant against this attack. However, resistance against tearing block corruption is absent, because the adversary can try as often as he likes to increment a counter by tearing. He does this by doing a restore from one block, and a transfer to the other, until the other block has a desired value. After then, the adversary can restore-transfer the desired value to the other blocks. This will result in the ADR bytes of the different blocks being equal. However, this is to be expected since the same procedure is used for recovering from accidental block corruption. Thus, strategy TNO-RU is resistant against a card-only key retrieval attack, but not against a tearing block corruption attack.

Strategy RU2 is resistant against both attacks. This follows from the fact that RU1 is resistant against tearing block corruption, and that RU2 is equal to RU1 except that it does not cater a key with write permissions. This resistance against both attacks comes at a price, however. RU2 cannot recover from all states in Table 4. Thus, card can accidentally get corrupted without the possibility to revitalise them. However, it remains possible to read all the data on the card, which will be useful in a restitution procedure.

These resistance properties are summarized in Table 5. To choose a strategy, one has to determine which attacks one considers plausible to exist. Then one takes the strategy which is resistant against those attacks. If one considers both attacks plausible, one has to either sacrifice functionality (by choosing RU2) or sacrifice security (by choosing either RU1 or TNO-RU).

### 3.2 Block Allocation Tables

While countersectors provide a transaction mechanism, we still need a way to define, for a given state of the countersector, which blocks of the card define the current *logical* state of the card.

The logical state of the card always includes the countersector. When a commit transaction is enacted, the card should be in a valid logical state, and after the commit transaction this should also be the case. Thus, the card needs the capacity to store two valid states simultaneously, and there is need for a convention which defines which blocks of the card make out the logical state, given the state of the countersector.

A complete transaction typically works like this: A reader authenticates to a card, reads out the current logical state, recovers it if needed, and it may verify the signature. Then it infers which memory blocks of the card are not part of the logical state. The new logical state is written to the free blocks, and then an atomic commit transaction is performed on the countersector. If somewhere during the whole process the card and the reader lose their connection, the reader has to re-authenticate and start all over again. The reader may not assume any persistence of the contents of free blocks over different connection sessions. However, the reader may not write to the card without the countersector being in a valid state.

This paragraph will elaborate on a few possible block allocation conventions. Ultimately and obviously, one convention should be chosen and stuck to. The list of conventions mentioned is is not exhaustive, but mentions some obvious and some efficient solutions.

Common to all conventions is the *oddity* of the counter which is stored in the countersector. The oddity can be observed by taking the least significant bit of the state counter. When the state counter is odd (resp. even), the logical card state is said to be ‘odd’ (resp. ‘even’). Moreover, without loss of generality we assume that the countersector is located in sector 1 (blocks 4-7) of a card, and that sector 0 (blocks 0-3) of the card is immutable (fixed).

**fifty-fifty** When the logical state is odd (even), all odd (even) sectors ( $\geq 2$ ) make up the logical card state. Preparing a logical state change involves writing a complete new logical state to the card. The storage capacity of a logical state is essentially a little under 50% of the physical storage capacity of the card.

**partial fifty-fifty** The same as fifty-fifty, but the convention does not apply to all sectors  $\geq 2$ , but only to a selection of those sectors. Typically a number of sectors is either left unused (because not that much memory is needed) or is used in another way (for example to store contents that should be immutable).

**allocation table** There is a compact representation which defines which data is stored in which blocks (or sectors) of the card. There are two reserved parts of memory on the card which store this metadata. We will call them *supernodes*. In even states, the even supernode is authoritative, in odd states the odd supernode is authoritative.

For example, the supernodes could each be 3 blocks large. Both supernodes offer 384 bits of metadata storage (3 blocks  $\times$  16 bytes  $\times$  8 bits). This allows the memory to be divided into 64 partitions which can be allocated individually. This can be seen by noting that it takes 6 bits to encode an integer between 0 and 63, and  $6 \times 64 = 384$ . The unit of allocation may then be the individual sector, but also the individual block. Note that blocks cannot be partially written, and therefore the block is the smallest possible allocation unit. As long as the smallest allocation unit is respected, many tricks and technologies generally used in file systems can be applied similarly to the Mifare Classic.

The allocation table takes up some storage space, but allows the logical card state to consume (considerably) more than 50% of the physical storage capacity of the card. Preparing a logical state change involves writing the changed parts to free blocks, writing an adjusted allocation table to the currently non-authoritative supernode.

The fifty-fifty convention is not particularly efficient. On the other hand, it is conceptually very simple and it does not restrict in any manner how much the state can change within one single transaction. The partial fifty-fifty reduces the fifty-fifty convention to only a limited area of the chip, which mitigates the overhead of making a transaction, but which also limits the storage capacity of a logical state.

The allocation table convention is a special case of the partial fifty-fifty convention. When state changes only change a moderate part of the logical state, this convention requires considerably fewer write operations than the other conventions. Moreover, it allows the logical state to take up more than

50% of the physical state, of course at the expense that state changes can no longer change 100% of the logical state. This can be seen by observing that if the logical state takes more than 50% of the memory of the card, there are insufficient free blocks to prepare a completely different new state.

### 3.3 Signature Suite

The standard way to create a cryptographic signature is to make a cryptographic hash of the message (in this case the logical card state) and then to sign this hash. On both the hash and the signature, there are some design choices to be made.

Since the Mifare Classic has only little memory, the memory footprint of these choices have to be taken into account. Also, computing the hash requires access to the logical card state, which may involve time-consuming read operations on the card. By using incremental hash functions or hash trees, storage capacity may be traded for transaction time. Using only one hash implies that a verification reader must have access to the complete logical card state. This may be undesirable, and using a hash tree may resolve this issue. We will discuss these issues one by one.

But first we will choose some security parameters. As this paper is an effort to make the best of Mifare Classic, we will choose the parameters such that the cryptographic guarantees are rather high. This will consume a particular amount of memory on the card. If one takes a more relaxed view to security, these parameters might be taken somewhat lower which clears up some storage space for application use.

The first parameter we choose is the bit length of the hash value. All other parameters are chosen to match the security level of the hash. We go for the option of 256 bits, which is considered “Good, generic application-independent recommendation,  $\approx 30$  years” (counting from the year 2006) [4]. This takes exactly 2 blocks of the Mifare Classic.

**Signature itself** There are two obvious options for the signature algorithm, namely RSA and DSA. An RSA signature is a number modulo  $n$  (where  $n = pq$ ). The cryptographic strength provided by a hash value of 256 bits is matched by an RSA key  $pq$  of 3248 bits [4]. A DSA signature is a tuple  $(r, s)$ , where  $r$  and  $s$  are numbers modulo  $q$ . The cryptographic strength provided by a hash value of 256 bits is matched by an DSA requires  $q$  to be 256 bits long[4].

Thus, given our choice for 256-bit hash values, a matching RSA signature requires 3248 bits (406 bytes) of storage. This takes almost 26 blocks of the card. However, a matching DSA signature requires 512 bits (64 bytes) of storage, which is only 4 blocks. Given the little storage capacity of the Mifare Classic, we opt for the signature algorithm with the smallest signature memory footprint, which is DSA.

As pointed out earlier, when no distinction between verification readers and state-modifying readers is needed, symmetric cryptography will suffice. In that case, the signature can simply be a MAC.

**Partial signature verification** A simple implementation of signed card states takes all block of the logical card state (except the signature itself), computes

the cryptographic hash over it, computes the signature over this hash and stores it in the logical card state. This requires that every verification reader and every state-modifying reader has read access rights to the complete logical card state, and exercises these rights exhaustively on every transaction.

There may be cases in which one does not give a reader access to the full logical state of the card, but where the reader should nevertheless verify up to some extent whether the card has a valid signature. Similarly, there may be cases in which a reader does have sufficient access rights, but for some reason or another should not exercise all these rights. (E.g. the read operations consume too much time.) In such cases, one can use a hash tree.

Without loss of generality, we divide the logical state of the card into a number of areas, each consisting of a number of blocks. One special area is the *root* area, which includes at least the countersector, and the (DSA) signature. The signature is a signature over the data in the root area. All readers have the Mifare Classic keys to read these sectors, and can trivially verify the signature, and with it the integrity of the root area.

Other areas we call *limited access* areas. The Mifare Classic keys to these areas may be restricted to only a subset of all genuine readers. To make the contents of a limited access area verifiable, the cryptographic hash of the limited access area has to be placed in *another* area. This other area may be the root area but also another verifiable limited access area. Thus, all limited access areas are linked to the root area either directly or chained via other limited access area. The Mifare Classic access keys must be distributed in such a manner that a reader which has access to a particular limited access area can verify the whole chain of limited access areas up to the root area.

Note that with the use of *incremental hash functions* [1, 2, 3], it is possible to update hash values without verifying them. This may be useful for situations in which there is insufficient time to read all the blocks that are required to actually verify their integrity. Using incremental hash functions essentially provides a way to change the state without verifying it, in such a manner that the new state only verifies if the old state would have verified.

**Saving space on hashes** Karsten Nohl has pointed out in private email that when the signature is a MAC, it may be possible to optimize on the number of bits required to store these ‘intermediate’ hash values which link limited access areas to the root area, possible via other areas. This crucially depends on three satisfiable properties: (1) the attacker not knowing the key used in the MAC, (2) every card has a unique MAC key, and (3) the card wears out. When the attacker does not know the MAC key, he essentially has to have the luck of guessing the correct MAC. The infrastructure may punish bad luck by permanently blocking the card. Using 32-bit MACs, the chance of luck is smaller than one in four billion. Alternatively, the attacker could wait to see the same MAC for two different values in the same block. For 32-bit MACs, that would take on average 65,000 transactions, which is already rather close to the write endurance of 100,000 transactions. Besides that, monitoring the card over 65,000 different transactions is may prove to be sufficiently impractical. Note that if the MAC key is share among different cards, it may be possible to distribute this process over different cards. Incorporating the UID of the card into the process, ideally by deriving the MAC key from the UID, makes this

impossible.

For the purposes of this paper, our main message is that it is possible to essentially trade storage space for security, and that it is possible to pre-emptively add redundant security to an RFID card such that if some security layers fail, others will remain. As Nohl rightfully points out, it may be possible to do this trade more economically than in the way sketched by us. However, we emphasize that such highly optimized trades must be analyzed thoroughly, as they may allow particular attack scenarios. By choosing the security parameters on the safe side, adhering to industry recommendations, such particular scenarios are pre-emptively ruled out.

## 4 Making Clones Distinguishable from Originals

In the previous section, we described a way which prevents attack scenarios in which the attacker possesses the attacked card, and modifies it at will. Of course, there is another avenue of attacks, which is cloning: The contents of a genuine card are read out, and copied onto a *clone host device*. The clone host device may be another (blank) Mifare Classic card, but it may also be another RFID card which is “Mifare Compatible”, or a generic RFID emulator such as the Ghost or the Proxmark3<sup>11</sup>, or an NFC chip which is mounted in a gadget like a mobile phone.

For a perfect clone, the clone host device must be indistinguishable from the original card, from the perspective of the genuine reader. Using side channel information, it is often possible to distinguish functionally equivalent devices. In a lab setting with specialised equipment, we have been to precisely measure, among other characteristics, the response time and field strength of clone host devices. The response time alone allowed us to distinguish between a genuine NXP Mifare Classic, a Fudan, and an NXP SmartMX (which is “Mifare Compatible”). Generic RFID emulators, can (at least in theory) be programmed to mimic the side channel characteristics of any desired clone host device as much as possible, including the original Mifare Classic.

Deployed production systems differ from a lab setting in a number of ways. A production reader typically consists of a non-modifiable NXP reader IC (which implements both CRYPTO1 and RFID radio modulation) and a programmable “firmware” IC which holds the application firmware. The NXP reader IC does not provide any means to collect side channel information, other than the response times of the reader IC itself. Theoretically it may be possible to program the firmware IC in such a way that it collects this timing information, but the typical firmware IC is not powerful enough to do this. Other side channel information, such as field strength, remains unknown to production readers. It is questionable whether it is wise to use side channel information in production systems. If one would use side channel information to reject possible clone host devices, this may have a serious negative impact on system robustness.

In this section we focus on information which off-the-shelf readers *do* have access to. This is the UID of the card used in the anti-collision-phase, the SAK (select acknowledge) value, and the (decrypted) responses to Mifare Classic commands (such as the contents of a block after it is requested). Note that using this information, one can also distinguish many clone host devices. The

---

<sup>11</sup>...

UID transmitted in the anti-collision typically leaks information about the manufacturer of the device. The Fudan mimics a genuine Mifare Classic 4K in this respect, but has a different memory layout which can be detected programatically.<sup>12</sup> Mifare Classics and licenced Mifare Compatible chips also have different manufacturer information in block 0 of the card. As such, we have been able to distinguish between the genuine, the unlicensed Fudan and licenced compatible versions of Mifare Classic while using proper sector keys but without using side channel information. Still, nothing beats an emulator.

Let us assume the reader cannot detect the type of clone host device. That is, the clone host device is either a Mifare Classic card of the same series, or it is a generic RFID emulator. What options are left for detecting clones?

Let us first consider the case where the clone host device is a (possibly counterfeited) Mifare Classic.

The obvious option for clone detection is using the UID of the card, which is used during the anti-collision and which is also stored in block 0 of the card. This UID cannot be changed.<sup>13</sup> A simple solution<sup>14</sup> is to store a MAC of the UID on the card. A reader which has the key  $K$  which is used to compute the MAC can detect whether the UID matches the MAC. As long as  $K$  is not known by the adversary, an adversary cannot fake such a MAC. A more elaborated solution is a cryptographic signature on the UID, for which readers only have the signature verification keys. If the transaction infrastructure of Section 3 is used, this can be established at no extra storage cost, by including block 0 of the card, which contains the UID, into the logical card state.

The problem cases left are UID-programmable cards and emulators. We present an approach which — roughly speaking — allows one to effectively detect and disable such clones. Precisely speaking, the approach *prevents* the creation of functional clones by making sure the adversary cannot gather sufficient data to know how a fully functional clone should behave. From here on, where we speak of a “clone” this colloquially means an “attempted clone”. An (attempted) clone can be distinguished from the original by a *clone test*: testing whether it behaves as it should; that it has the data which it could not have collected from eavesdropping.

**clone tests** The general idea to distinguishing clones from originals is the following. The original card is prepared with special data on specially reserved sectors of the card. During operation, these sectors are left untouched. In particular, any keys required to read these sectors are not used, and therefore an adversary cannot recover those keys, nor recover the contents of the blocks protected by these keys.

At some moment a reader may want to test whether the card he is interacting with is a clone or not. At that moment, the reader will try to authenticate for one

---

<sup>12</sup>For the memory lay-out differences, see Table 1. We have also been able to distinguish between the counterfeit and the genuine Mifare Classic without engaging in authentication, by sending particular invalid commands to the card. The two versions react differently. It is a matter of definitions whether one considers this side channel information. However, standard NXP reader ICs do not facilitate this test.

<sup>13</sup>There have been rumors of unlicensed counterfeit Mifare Classic cards for which the UID are programmable, but we have not been able to confirm these reports.

<sup>14</sup>The simplest solution is of course a whitelist of used UIDs, known to all readers. However this is in general not practical.

of these sectors which has not yet been accessed. Upon successful authentication, the reader can ask for the specially prepared data.

This approach offers two possible tests which an original card will always pass, but which a clone may fail to pass:

**key test** An original knows the one key that should authenticate, a clone does not. Thus, a clone will have to bet whether the presented key is correct.<sup>15</sup>

**data test** An original card holds the specially prepared data, a clone does not. Thus, a clone will have to present some fake special data, and hope it is correct.

The *key test* has some limitations. If the wrong key is presented to a genuine Mifare Classic the card will enter in the halted state, in this state the card no longer interacts with the reader. Then, the card can only be interacted with after a new anti-collision and select procedure, and the reader cannot detect whether it is talking to the same device or not. (Remember, we are assuming emulators are around.) A reader which willingly tries to authenticate with the wrong key to detect whether the card actually halts, will have to accept that the original *will* actually halt. Thus, the *key test* is only practical at the end of a reader-card interaction: a complete transaction is performed as if the card is an original, and at the end it is tested whether the card is in fact an original. In our solutions, we will not use key tests, but only data tests.

The *data test* has fewer limitations. The data must be stored over the blocks of the corresponding sector. In the 64-byte sectors, which have two keys each, three blocks are available, each 128 bits (16 bytes) large. Particular access bit configurations allow one sector to facilitate two data tests. The first test uses key A, which gives access to one or two blocks of the sector. The second test then uses key B, which gives access to the full sector. However, performing the test with key B *before* the test with key A will render the test with key A useless, as key B will expose the specially prepared data of test with key A.

The trailer access bits  $C_1C_2C_3$  must be two-key, allowing key change, i.e. 011, 101 or 100. The blocks not readable by key A must be *dead* with respect to key A, i.e. in configuration 011, 101 or 111. Care must be taken that the sector can be reprogrammed using key B, which is not allowed in all combinations of these access bits. A simple configuration which allows full reprogramming of the sector but still provides the required functionality is where the trailer access bits are 011, the blocks *not* readable by A are in configuration 011, and the blocks *readable* by key A are configured either 000, 110 or 100.

## 4.1 Online vs. semi-offline testing

The application setting of a card dictates what policies for clone detection are acceptable and which not. For example, in access control to military facilities, where it is extremely important to keep intruders out, it will be unacceptable

---

<sup>15</sup>Of course for making any bet as to whether the presented key is correct, the clone does not necessarily need to know the correct key. Nevertheless, the authentication protocol leaks information which allows the clone to efficiently compute a set of  $2^{16}$  keys of which it can be sure that it contains the presented key. For sessions where multiple sectors are authenticated, the second and subsequent authentications leak that much information that the set has only  $2^6$  (i.e. 64) elements [5]. Subsequent commands from the reader will give sufficient redundancy to select the used key from this little set.

if clones cannot be detected immediately. Some moderate amount of hassle for legitimate users may be acceptable for the protection of such high-security facilities. In other applications, such as public transport ticketing, the residual risk of an adversary taking an occasional free ride on the subway may be acceptable, while it is important that the legitimate user has as little inconveniences as possible.

For both application areas we present a solution. For the access control application (detect clones immediately) we require the readers to be online, that is, always in direct contact with the back office. All the currently deployed RFID access control systems we know of consist of online readers. Our online solution is simple and effective. For the public transport ticketing application (detect clones eventually) the readers may be any mixture of online and semi-offline readers. Semi-offline readers synchronise with the back office at specified intervals. All the currently deployed RFID public transport ticketing systems we know of have a mixture of online and semi-offline readers.

A semi-offline reader is online every now and then, with a specified minimum like “once a day”. A typical example of a semi-offline reader is a reader which is mounted in a public transport bus, it connects to the network and synchronises with the back office every evening at the bus depot. A semi-offline reader has sufficient memory to store all card interactions during an offline period, and to store greylists and blacklists of a moderate size. Every time a semi-offline reader synchronises with the back office, it sends all card interactions since the last synchronisation to the back office, and updates its greylists and blacklists. Note that online readers are a special case of semi-offline readers; online can synchronise with every card interaction.

Our solution can reliably detect attempted clones, and disable the clones without disabling the original card.

**online testing** The online setting is, compared to the semi-offline setting, very simple. When a reader encounters a card, it gets the card ID from the card and sends it to the back office. The back office keeps track of which tests have been performed on that card, and sends back to the reader the test number of a so-far not unused test. The back office marks the test as used, and the reader performs the test. If the card passes the test, the reader judges that the card is not a clone.

If the card fails the test, the reader will judge that the presented card is an attempted clone. What to do in these cases is a matter of policy: nets might be catapulted onto the person holding the clone to the reader in order to capture the offender, a camera may take a picture of the offender, security personnel may be directed to the spot. Just refusing access may in cases also be a sufficient solution. The fact that a particular card ID has attempted clones may be reported by the reader to the back office. This information can be used for monitoring and procedural purposes.

## 4.2 semi-offline testing

In the semi-offline case, clone detection is not as straightforward. Moreover, in the semi-offline case it cannot be prevented that a clone *never* enters the system. However, clones entering the system can be detected after the act, and these clones can be blacklisted without also blacklisting the original card. The latter

is a very important contribution. In all deployed clone-detection systems known to us<sup>16</sup>, the original card will be blacklisted as well, resulting in a considerable amount of hassle for the legitimate card owner.

The rough idea is the following. The semi-offline setting detects and disables clones in two separate phases. In the first phase, transactions are monitored offline to detect which card IDs show exhibit suspicious patterns. These card IDs are then *greylisted*, which puts them in the second phase. When a reader encounters a card with a greylisted card ID, it will perform a test for which it can tell on the spot whether the presented card is a clone or not. When a reader encounters a greylisted card which passes the test, it will *remint* the card, that is, change its card ID. The old card ID will be *blacklisted*. Greylists and blacklists are synchronised with the back office every time a reader is online.

Every card has a card ID, which consists of two parts. The first part (*cID-fixed*) is immutable, typically the hardware serial number of the card. The second part (*cID-flex*) is mutable. A *valid* card ID is a combination *cID-fixed*, *cID-flex* which satisfies a cryptographic property which can only be verified with a particular verification key. Given *cID-fixed*, to generate a valid *cID-fixed* requires a generation key. Using symmetric cryptography, *cID-flex* can be a random number plus a MAC over *cID-fixed* and that random number; and anyone who can verify card IDs can create new card IDs. Using asymmetric cryptography, *cID-flex* can be a random number plus a signature over *cID-fixed* and that random number; and anyone who can verify card IDs can create new card IDs. The parameters are chosen such that it is infeasible to guess which card IDs will be valid.

Every card has a *hello counter*. When a reader encounters a card, it reads the hello counter and increments the counter by one. The reader also logs the hello counter, together with the card ID in its card interactions which are synchronised with the back office. The value stored in the countersector, described in Section 3.1 may be used for this purpose<sup>17</sup>, but that will prevent verification readers (Section 3) from assisting in clone detection.

The back office collects for every card ID the hello counter values. If everything is okay, these values are all unique and strictly increase over time. When a card is cloned, and both the original and the clone are in use, the back office will receive card interactions which constitute a sequence which does not strictly increase over time. This can be easily detected. How quickly the back office detects this depends on how intensively the original card and the clone are used, and how often the readers synchronise with the back office.<sup>18</sup> When such an anomaly is detected, the card is *greylisted*, and this greylist is then distributed to all readers. The greylist record also contains a new card ID, and the number of the first untouched data test of the card. The new card ID is also put on a *whitelist* which is kept by the backoffice.

When a reader encounters a greylisted card, it will perform the so-far untouched data test. If the card passes the test, it must be the original card. The

---

<sup>16</sup>e.g. The systems used in the London Oyster Card and the Dutch OV-chipkaart.

<sup>17</sup>In that case, the countersector value needs to be negated to get a monotonically increasing counter.

<sup>18</sup>Of course, a rather small tolerance should be allowed for the clock differences between the readers. This does however not change the analysis. For clarity, we have omitted this from our pseudocode.

```

function detect-genuine(greylist, clone-blacklist, card-blacklist)  $\rightarrow$  genuine
  cID-fixed  $\leftarrow$  get-cID-fixed(); /* read */
  cID-flex  $\leftarrow$  get-cID-flex(); /* read */
  hello-counter  $\leftarrow$  get-hello-counter(); /* read */
  if (not valid-cID(cID-fixed, cID-flex)) then
    log(“invalid-cID”, cID-fixed, cID-flex, hello-counter);
    return false; /* invalid cID found */
  if [cID-fixed, cID-flex] in clone-blacklist then
    log(“blacklisted-clone”, cID-fixed, cID-flex, hello-counter);
    return false; /* blacklisted clone found */
  if [cID-fixed] in card-blacklist then
    log(“blacklisted-card”, cID-fixed, cID-flex, hello-counter);
    return false; /* blacklisted card found */
  set-hello-counter(hello-counter + 1); /* write */
  if [cID-fixed, cID-flex] in greylist then /* greylisted card found */
    test-nr  $\leftarrow$  get-test-nr([cID-fixed, cID-flex], greylist);
    passed-first  $\leftarrow$  perform-data-test(test-number); /* read */
    if (not passed-first) then
      log(“failed-first”, cID-fixed, cID-flex, hello-counter);
      return false; /* first test failed */
    else /* first test successful */
      try
        passed-second  $\leftarrow$  perform-data-test(test-nr + 1); /* read */
      catch lost-card-error
        passed-second  $\leftarrow$  false;
      if (not passed-second) then
        log(“failed-second”, cID-fixed, cID-flex, hello-counter);
        return false; /* second test failed */
      else /* second test successful */
        cID-flex  $\leftarrow$  get-new-cID-flex([cID-fixed, cID-flex], greylist);
        remind(cID-flex); /* write */
        overwrite-data-tests(test-nr, test-nr + 1); /* write */
    log(“normal”, cID-fixed, cID-flex, hello-counter);
    return true; /* genuine card found */

```

Figure 9: Pseudocode for semi-offline readers. All log messages are annotated with the time of the card interaction, and are sent to the back office the next time the reader synchronises. All writes are atomic, using a transaction mechanism like the one described in Section 3. All lines annotated with “read” or “write” may access the card, and may throw an exception if the card is removed from the reader field. An exception causes the function to be aborted and the exception is passed to the calling function.

```

function process-log-entry(time, text, cID-fixed, cID-flex, hello-counter,
                          whitelist, greylist, clone-blacklist, card-blacklist)
if text = “normal” then
  select [b-time’, cID-fixed’, cID-flex’, b-hello-counter’] from normal-log
    where (cID-fixed’ = cID-fixed) and (cID-flex’ = cID-flex)
    and (b-time’ < time) and (b-time’ is maximal);
  select [a-time’, cID-fixed’, cID-flex’, a-hello-counter’] from normal-log
    where (cID-fixed’ = cID-fixed) and (cID-flex’ = cID-flex)
    and (a-time’ ≥ time) and (a-time’ is minimal);
  if (exists(b-hello-counter’) and (b-hello-counter’ ≥ hello-counter)) or
    (exists(a-hello-counter’) and (a-hello-counter’ ≤ hello-counter)) then
    /* phase one anomaly detected: hello counter out of sync! */
    if [cID-fixed, cID-flex] not yet in greylist then
      /* this is the first anomaly for this particular cID-flex */
      new-test-nr ← get-next-unused-test-nr(cID-fixed);
      new-cID-flex ← generate-new-cID-flex(cID-fixed);
      add(whitelist, ⟨[cID-fixed, new-cID-flex], cID-flex, new-test-nr⟩);
      add(greylist, ⟨[cID-fixed, cID-flex], new-cID-flex, new-test-nr⟩);
      // possibly some other administrative actions ...
    insert into normal-log value
      [time, cID-fixed, old-cID-flex, hello-counter];
    if [cID-fixed, cID-flex] in whitelist then
      /* reminded card detected */
      test-nr ← get-test-nr([cID-fixed, cID-flex], whitelist);
      set-processed-test-nr(cID-fixed, test-nr + 1);
      old-cID-flex ← get-old-cID-flex([cID-fixed, cID-flex], whitelist);
      remove(whitelist, ⟨[cID-fixed, cID-flex⟩]);
      remove(greylist, ⟨[cID-fixed, old-cID-flex⟩]);
      add(clone-blacklist, ⟨[cID-fixed, old-cID-flex⟩]);
      // possibly some other administrative actions ...
    if (text = “failed-second”) then
      /* partially successful clone test detected */
      new-cID-flex ← get-new-cID-flex([cID-fixed, cID-flex], greylist);
      remove(whitelist, ⟨[cID-fixed, new-cID-flex⟩]);
      remove(greylist, ⟨[cID-fixed, cID-flex⟩]);
      add(card-blacklist, ⟨[cID-fixed⟩]);
      // some other administrative actions ...
    if (text = “invalid-cID”) or (text = “blacklisted-clone”) or
      (text = “failed-first”) or (text = “blacklisted-card”) then
      // possibly perform additional exception handling ...
      // for example retrieve camera footage of the relevant reader at time time

```

Figure 10: Pseudocode for the back office.

reader will then *remint* the card, i.e. change the card ID. From this transaction onwards, the card interactions caused by this card will contain the new card ID, which is also on the backoffice's whitelist.

When the card interactions of a reminted card are processed by the backoffice, the backoffice can detect the actual reminting of the card because the new card ID is on the whitelist. When this happens, the backoffice will update various lists: the old card ID can be removed from the greylist and put in a blacklist (the *clone-blacklist*). As this has to be done only once, the new card ID can be removed from the whitelist. The adjusted greylist and the clone-blacklist are then distributed to all readers.

When a reader encounters a card which is blacklisted or a card with an invalid card ID, it will refuse the card.

Readers may of course, in anticipation of updates from the backoffice, update their locally held greylists and blacklists as a result of local events, or local events of other readers it can contact while being out of touch with the backoffice.

Of course, in practical situations with huge numbers of deployed readers, some readers will fail to synchronise or may go defect. This has little impact on the overall system. As long as the vast majority of log messages is processed, the detection of clones will work as advertised. Moreover, the reminting of a card is detected also if the reader that performed the reminting dies before synchronising, because the reminted card will leave a trace at other readers. Obviously, a reader that has stopped synchronising cannot access up-to-date blacklists.

When the genuineness test of a reader fails, the reader can be sure that something seriously went wrong.<sup>19</sup> This should at least result in access being denied. Depending on company policy, this may also trigger the reader to collect auxiliary information such as CCTV footage. Such trigger-initiated footage will likely contain images of an offender. Other footage may be deleted, or not be produced in the first place. Such policies dramatically minimise the storage of CCTV footage of innocent users of the system, while producing footage of reasonably suspect people.

If this were the whole procedure, there would be a possibility for the adversary to get his clone reminted and the original card blacklisted. The scenario is this: the attacker sniffs transactions of the original card, creates a clone and uses this clone to trigger greylisting of the card. When the card is greylisted, the adversary immediately goes to a reader to retrieve the Crypto-1 key for the data test. With this key, the adversary goes to the original card and reads out the data test block and the vandalises this block on the original card. Then the adversary goes to a gate with his clone, and presents the correct data on the data test. The clone will be reminted. In this attack, timing is crucial: if the legitimate card owner remints first, then the adversary is out of luck.

Though it may be impractical to find back a *particular* sniffed card to read its data test block, in general it can be rather easy to collect this data for a moderate number of cards. Consider the public transport ticketing case. Commuters typically make the same trip very working day, entering and leaving through the same stations, possibly through the very same gates. When such a

---

<sup>19</sup>If only the card is removed from the reader field, the test will throw an exception instead of returning false.

gate is illegally extended with an eavesdropping device and an extra reader, the eavesdropping device can gather the keys and data for the “normal blocks” of the card. This data is then used to trigger greylisting of the sniffed cards. The reader reads out the card ID of every card approaching the genuine reader. If the card ID is on the hit list of the malicious reader, it will read out the data test block and vandalise it directly afterwards. For commuters the chance of falling victim to this kind of an attack is considerable, or at least more than negligible.

The existence of this attack scenario lies in the following combination of facts. Firstly, eavesdropping two failed authentication attempts by a genuine reader for a particular sector are enough to recover the key. Secondly, the original card should always be given a chance to pass the genuinity test, no matter how often the adversary has tried to do so. Now if the genuineness test only uses *one* data test and thus only one key, this key will be easily recovered by the adversary.

If however the genuineness test uses *two* data tests, and the second test will only be performed after successful completion of the first data test, and *only once*, the second key will not be exposed. This immediately suggests how to thwart this attack scenario. Two data tests are used. If a card passes the first test, it must immediately pass the second test. If it fails the second test, for whatever reason, the card (both the original and the clone) will be blacklisted (using *card-blacklist*).<sup>20</sup>

An adversary may cause a legitimate card to be blacklisted, but that will always result in the clone also being blacklisted. Note that this does not open a functionally new way for an attacker to cripple a genuine card; an attacker can always cripple a legitimate card by vandalising an often-read block of which he can easily obtain the write key.

Figures 9 and 10 give pseudocode that describes the presented solution in detail.

The fact that the second data test may only be performed only once means that there is a critical part of the transaction that may not be interrupted, between starting the first data test and reminting the card. Failures caused by legitimate users taking their card out of the field in this small time frame will cause the card to be erroneously blacklisted. It should be appreciated that the chance that a legitimate card must execute the critical section of the protocol depends on the fraud levels, but will nevertheless be smallish (say, way below 1%). The chance that the transaction is interrupted in the critical section depends on user behaviour, but can also be assumed to be rather small. One of the following three approaches (which for clarity are not reflected in the pseudocode) can be taken to address this.

**redirect** A simple approach is that if a reader encounters a greylisted card, it instructs the card holder to go a special vending machine where the transaction can be performed in a more controlled environment (i.e. a Faraday cage).

**accept the risk** An even simpler approach is to simply take the risk that the

---

<sup>20</sup>Note that, in contrast with the rest of our approach, this particular blacklisting action depends on one single log message which must not get lost between the reader and the back-office.

protocol is interrupted. The chance is very small indeed, the inflicted hassle is rather big: the card holder can no longer travel.

**accept, then redirect** A more complex approach is twofold. One, we simply take the risk that the transaction is interrupted accidentally. This chance is in fact rather small, though not completely negligible. Two, if the card happens to be blacklisted on the spot, the card holder is instructed to go to a vending machine. At the vending machine, the card will undergo a *third* data test in a controlled environment (i.e. a Faraday cage). If the card passes the test, the card will be reminted and the card blacklisting will be limited to clone blacklisting. As long everything goes well, there is simply no hassle. If the critical section is interrupted, the card holder has some hassle.

The “redirect” and the “accept-then redirect” approaches require an infrastructure of vending machines. In the latter approach, the volume of interruption-caused transactions for the vending machines will be orders of magnitude smaller than in the former approach. It may be that the “accept-then redirect” case causes this few exceptions that it is acceptable to refrain from having vending machines available on every station or bus stop.

This procedure effectively detects and disables clones while keeping the originals intact. However, that does not mean that an adversary cannot use clones to his advantage, as the process takes some time and leaves a window of opportunity.

For one, it takes some time before a deployed clone is disabled; if the synchronisation takes place every night, an adversary can use a clone effectively for up to two days. If the original card is not used for some while after its transaction is eavesdropped, detection takes the same while of time. In a public transport ticketing situation, when an adversary eavesdrops card transactions on a massive scale, this window of opportunity may still be economically interesting.<sup>21</sup>

On the other hand, an adversary may purchase a card, clone this card and use it until it is disabled. Then he simply clones his original card again, which can be repeated any number of times. Of course, this can be detected in the back office, as the original card is re-issued a new card ID many times. When the re-issue count reaches a certain threshold, it may be assumed that the card is being using in this unintended manner. Then the card ID of the original can be blacklisted as well. If the card holder is not anonymous, he may even be asked to give an explanation of the course of actions.

### 4.3 Initialising data tests

**initialisation environment** An essential requirement for the effectiveness of data tests is that the data to be presented in the tests is kept secret until tested. Thus, read keys should never be exposed. When a card is initialised before it is issued to the user, this can be guaranteed rather easily. However, only a limited number of sectors can be dedicated to data tests. After a few reminting

---

<sup>21</sup>Note that we do not make any statement as to *whether* this enables a criminal business case, we only state that it *may*.

operations, the card will eventually run out of unused data tests. It is possible to overwrite the sectors with new data tests, but this cannot be done in a public environment, as the keys used to overwrite will be leaked by doing so.

This can be addressed in essentially three different ways. All solutions work in the public transport application, the last one also works in the access control application.

1. The simplest solution can be applied if there is a policy to block a card after a particular number of reminding operations. The number of data tests is chosen to be twice the threshold of allowed reminds. When the data tests run out, the card is blocked, and thus there is no need to overwrite the data tests with new ones. This of course only works when the threshold is not too high, due to space considerations.
2. Another solution can be applied if the user is known by name and address. When the number of unused data tests drops below a particular threshold, the card issuer pre-emptively creates a new card for the user and sends it to the user. When the user starts using the new card, the old one is no longer needed and can be disabled (i.e. blacklisted).
3. The last solution can also be applied in access control applications. There are special devices for re-initialising the data tests, where the users can go to to have their tests re-initialised. These devices require the card to be inserted completely, and will overwrite the data tests within a Faraday cage. As damaging the Faraday cage may make the security collapse, these devices should be in a guarded environment, at least within the security perimeter enforced by the access control system. Moreover, it should be almost impossible to transport these devices, or if that is not possible the devices should stop functioning if moved from the deployment location. This can be established by making the devices online, and have them stop functioning if offline. One way to achieve this is that the devices do retrieve the actual keys using a backoffice connection; moving the device should then break backoffice connectivity.

In an access control setting, data tests need to be re-initialised frequently. Depending on how often the user crosses a security perimeter, this may be typically once a week or so. Card readers may warn the user if his card is almost out of tests, urging them to go to a re-initialisation device.

**verification environment** When a reader has to perform a data test, the reader has to have a means to verify whether the presented data is correct. However, it has to be prevented that a reader can be used as an oracle which can produce the data that should be presented in a data test.

In the setting with tamper-resistant online readers, this can be done easily. The reader contacts the backoffice and the backoffice sends the required data via an encrypted tunnel between the reader and the backoffice. The backoffice will tag the data test as performed. The reader will only open the door if the tag passes the data test without further ado. If the same tag is presented again, it will contact the backoffice for the next data test.

In case the readers are or not always online, other solutions may be used. One is to give every reader a master key which it used to derive the data test

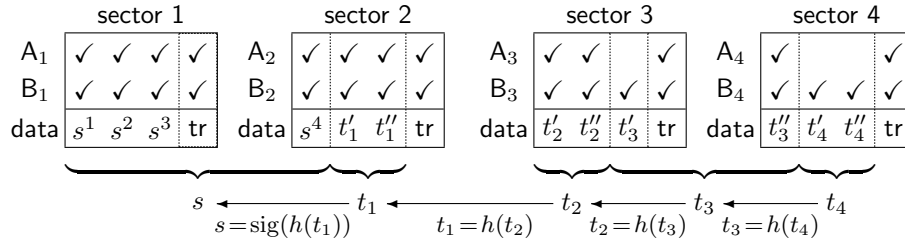


Figure 11: A data test chain mapped to sectors. For every sector, it is given which of its keys give read access to which blocks of the sector. For every block, its contents are given. In this figure, the chain is 4 tests long, and anchored with signature  $s$ .

data from, another is that the backoffice distributes the data test data jointly with the greylist. However, both these solutions require the reader to be tamper proof. If the reader is not, it can be turned into an data test data oracle, which will make the security collapse.

A reader which is not always online, and which may be not sufficiently tamper resistant, should be able to verify the correctness of data test data, without being able to produce the data test data. There is in fact a reasonably simple solution for this: the card has to present some unguessable data which carries a public-key signature from the card issuer, i.e. a random value and a signature over this random value.

Signatures are somewhat space-consuming, and it is possible to save considerably on storage space if multiple tests can be anchored using one single signature. In fact, there is a not too complicated solution for this, which is essentially a hash tree consisting of one long branch, and a public key signature over the root node of the tree. When a card is initialised, it is chosen how many data tests should be on the card. Let this number be  $n$ . Data test data  $t_n$  is chosen randomly, and for all  $i < n$   $t_i$  is set to  $h(t_{i+1})$ . The public key signature over  $t_0$  is computed, and this signature and  $t_1$  through  $t_n$  are stored on the card. The reader performing data test  $j$  reads out  $t_j$ , computes  $t_0$  by taking the hash of  $t_j$   $j$  times, and verifies whether the signature on the card is a valid signature over the thus computed  $t_0$ . The data tests have to be performed in ascending order, as performing test  $j$  exposes all the information to pass any test up to test  $j$ .

Figure 11 shows such a hash chain which can be used for four data tests. For test 1, the reader has to have keys  $A_1$  and  $A_2$ , and read out  $s$  and  $t_1$ . For test 2, the reader still needs these keys to read out  $s$ , and needs  $A_3$  to read out  $t_2$ . In test 3, keys  $B_3$  and  $A_4$  are needed to read out  $t_3$ , and in test 4  $B_4$  is needed to read  $t_4$ . The read rights of the keys are chosen with care such that keys for test  $j$  do not offer rights to read  $t_{j+1}$ . The chain can easily be extended to span more sectors and to facilitate more tests. As the test number increases, the number of computations to be performed by the reader increase linearly. However, it is reasonable to assume the reader has a moderate amount of computing power. The number of read operations does not change as the test number increases. The number of required authentications is at most four (two to read the signature and two to read the  $t$  value, both of which are

divided over to sectors each).

## 5 Key list

In this paper, a large number of keys have been introduced. Using them all in the prescribed manner may improve Mifare Classic security. Mixing up the keys may lead to disasters. Therefore, here we list and summarize all keys that are proposed in this paper.

Where keys should be diversified, diversification should be done on both card UID, sector number, and whether it is key A or B.

**Mifare sector keys (CRYPTO1)** Exposure of these keys of a card allow an attacker to cripple that card. These keys are used regularly and can be intercepted, therefore it does not make sense to hide them too obsessively. To make a large-scale denial-of-service by vandalism slightly more difficult, the Mifare sector keys should be diversified from a master key.

**Mifare master key (MASTER)** This key is given to all readers in the system. Exposure of this key facilitates denial of service, but does not facilitate fraud. Seen from a criminal perspective, this key might allow the “blackmailing” business case, but not the “fraud” business case.

**Mifare sector write keys for the countersector (optional, CRYPTO1)** This key allows changing the counter sector into any desired state. This is needed to restore the countersector when it has been corrupted by taking the card out of the reader field at exactly the wrong moment. This key will only be used in a controlled environment, at card issuance and at special readers where it cannot be intercepted (i.e. a Faraday cage). The re-initialisation readers may be used given the extra functionality to recover corrupted countersectors. Exposure of this key will allow state restoration attack on a single card. To make large-scale state-restoration more difficult the key should be diversified from a master key.

**Mifare master key for write access to countersectors (optional, MASTER)** This key should be given a very strict regime, as leaking it would enable massive state restoration attacks. Besides during card initialisation, it is only needed in special cases. Readers recovering counter sectors might be given this key, but it is wiser to keep this key in the backoffice and have recovering readers request diversified samples on demand. This enables monitoring the usage of this key. Seen from a criminal perspective, this key allows the “fraud by state restoration” business case.

**Mifare sector keys for data tests (CRYPTO1)** This key allows one to read the data test data of a card. Exposure of this key before an actual data test enables the bypassing of clone tests of a particular card. Therefore, this key must be diversified per card and per sector. Moreover, it is wise to diversify this key from another key than the Mifare master key.

**Mifare master key for data tests (MASTER)** This key should be given a very strict regime, as leaking it would enable massive cloning without being able to distinguish between originals and clones. This master key is present

only in the backoffice. In the access control application case, the reader contacts the backoffice to request a particular diversified key upon every test. In the public transport application case, when a card is greylisted, the appropriate read keys are distributed to the readers together with the greylist. Readers doing re-initialisation might be given this key, but it is wiser to keep this key in the backoffice and have re-initialising readers request diversified samples on demand. This enables monitoring the usage of this key.

**Card state public key (DSA-public)** This key allows one to verify the validity of a card state. This key may be distributed freely.

**Card state test private key (DSA-private)** This key allows one to change the card state. It is needed by all state-modifying readers. Exposure of this key will allow attackers to change the state at will, which notable includes changing stored (monetary) values to any desired value.

**Card state symmetric key** If there is no need for a special class of verification readers, the card state may be “signed” with a symmetric key (e.g. a MAC). In case the approach of Karsten Nohl (given at the end of Section 3.3) to save storage space on the hash tree is used, this key can be used for construction and verification of the MACs. Exposure of this key will allow attackers to change the state at will, which notable includes changing stored (monetary) values to any desired value. This key will be needed both for verification and modification of card states.

**Data test public key (DSA-public)** This key allows one to verify the signature involved in a clone test. This key may be distributed freely.

**Data test private key (DSA-private)** This key is needed when data tests are initialised. Exposure of this key will enable massive cloning without being able to distinguish between originals and clones. Readers doing re-initialisation might be given this key, but it is wiser to keep this key in the backoffice and have re-initialising readers request new signed data test data on demand. This enables monitoring the usage of this key.

## 6 Conclusion

The Mifare Classic has a number of security features built in. Some turned out to be flawed. Some features, which are not particularly advertised as security features, can be used as a building block for additional security mechanisms. This document describes such additional security mechanisms.

As a result, it is possible to prevent successful state restoration attacks, and to prevent cloning attacks. That is, we currently believe these countermeasures are effective. However, they are provided “as is”. We welcome feedback, decent peer review and further research is required. Crucially, though obviously, our countermeasures will not be effective if the features which are used as building blocks work can be circumvented. New attacks might appear which could make the countermeasures described in this document obsolete. Of course, similar reasoning applies to any kind of RFID card.

System owners will have to decide for themselves whether they should implement these countermeasures. It may be that migrating to a wholly different card is just as cumbersome as implementing these countermeasures. However, one may want to implement these or similar countermeasures also on a newly selected card. That will pre-emptively offer extra security layers in case the cryptography of the newly selected card will fail at some point in the future.

## 6.1 Acknowledgements

This document has seen many revisions, we started working on it when the results of [5] were not yet fully finalised. The ESORICS paper [5] was highly influential. Due to a court order, it was not possible to distribute draft versions of [5] until the ESORICS conference itself. Drafts of the current paper have been distributed to NXP (which had detailed knowledge of the ESORICS paper) and to TNO and Karsten Nohl (who were not in a position to have access to the ESORICS drafts).

Karsten Nohl, NXP and TNO have given very valuable feedback, and I thank them for it. In particular the undocumented feature given at the end of Section 2.2, the method to save storage space on hashes given at the end of Section 3.3 from Karsten Nohl, and the TNO-RU strategy given in Section 3.1 from TNO were welcome contributions. I thank NXP for patiently asking clarifications on the drafts, and for patiently answering our questions.

I am greatly indebted to my colleagues, with whom I experienced the adventure of getting to grips with Mifare Classic. Their results, and the discussions with them have helped this paper to be what it is. They are: Flavio Garcia, Jaap-Henk Hoepman, Bart Jacobs, Ravindra Kali, Vinesh Kali, Gerhard de Koning Gans, Ruben Muijers, Peter van Rossum, Roel Verdult and Ronny Wichers Schreur.

I believe that security by obscurity is bad. The details of possible attacks have to be public knowledge, and the details of possible countermeasures as well. Without insight in the attacks, it is difficult to evaluate the effectiveness of countermeasures. Restraining access to this information hampers progress in scientific security research. Such scientific research is needed to advance the application of security products in society which are actually secure. It is bad when vulnerable products are around for a long time; non-ethical knowledgeable people will exploit these vulnerabilities without disclosing that these vulnerabilities exist. Apparently disclosure of vulnerabilities is needed to convince the responsible parties to migrate to better solutions.

## References

- [1] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: the case of hashing and signing. In Y.G. Desmedt, editor, *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, Berlin, 1994. Springer-Verlag.
- [2] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography with application to virus protection. In *Proceedings of the 27th Annual Symposium on the Theory of Computing*. ACM, 1995.

- [3] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *Advances in Cryptology- EUROCRYPT 97 Proceedings*, volume 1233. Springer-Verlag, 1997.
- [4] ECRYPT. Yearly report on algorithms and key sizes (2006), January 2007.
- [5] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling mifare classic. In *Proceedings of ESORICS 2008, 13th European Symposium On Research In Computer Security*, pages 97–114, Malaga, Spain, 2008.
- [6] NXP Semiconductors. MF1ICS70 functional specification. online on mifare.net, January 2008.