UNIVERSITY OF CAMBRIDGE

Daniel Chatfield
Robinson College
dc584

Computer Science Tripos — Part II, 2016

Security Countermeasures for MIFARE Classic RFID Cards

Supervised by

Dr. Markus Kuhn

Proforma

Name:	**Daniel Chatfield**
College:	**Robinson College**
Project Title:	**Security Countermeasures for MIFARE Classic RFID cards**
Examination:	**Computer Science Tripos — Part II, 2016**
Word Count:	11,995*
Project Originator:	Dr Markus Kuhn
Supervisor:	Dr Markus Kuhn

Original Aim of the Project

The original aim of the project, as set out in the proposal, was to develop countermeasures that mitigated the risk of attack against MIFARE Classic cards. The proposed countermeasures consisted of a digital signature library and a gossip protocol for distributing revoked UIDs.

I also planned to write a MIFARE library which would be used by the countermeasures to communicate with MIFARE Classic cards.

Work Completed

The project encompasses the original aims. I have produced a MIFARE library, digital signature library and a revocation gossip protocol.

In addition to my original aims, I have also devised several additional countermeasures and a comprehensive simulation suite for simulating networks of readers and cards.

Special Difficulties

None

*Computed using `texcount -total -template=SUM *.tex`

Declaration of Originality

I, Daniel Chatfield of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

SIGNED

DATE

Contents

Chapter 1

Introduction

The MIFARE Classic card is an NFC* based contactless smart-card with between 1 KB and 4 KB of protected memory. The cards are used for a wide range of applications, including micro-payments, ticketing, public transport and access control. The card is used within the University of Cambridge for access control to buildings as well as payment for meals at most colleges. To read or write data to the card an NFC reader must first authenticate with a secret key. Researchers have uncovered weaknesses in the card that allow an attacker to extract the secret keys and thus bypass the authentication. Given the range of applications the impact of this vulnerability is large.

This project explores various countermeasures that an organisation can use to mitigate the risk of attack until they can move to a more secure card. The project consists of:

- A library for interacting with MIFARE cards
- A simulation suite for simulating a network of readers and cards
- A library for reading and writing digitally signed data to MIFARE cards
- A gossip protocol for distributing a list of revoked cards to offline readers (those without a network connection)

1.1 Motivation

Since the vulnerabilities were made public in 2008, some organisations with large deployments of MIFARE Classic cards have upgraded to cards without known exploits. Transport for London started replacing the MIFARE Classic based Oyster cards with MIFARE Desfire ones in 2010 to control fraud.

*Near Field Communication — a set of communication protocols that enable two devices within close proximity of each other to communicate

Many organisations, including the University of Cambridge, have yet to upgrade and still rely on MIFARE Classic cards for authentication and access control. More secure cards can't be issued until all the readers are upgraded to support them. For an organisation such as Cambridge, this is a fairly involved process which could take a long time. Cambridge consists of hundreds of suborganisations including departments and colleges, each of which have their own reader deployments.

Existing research[1][2][3] has focussed on finding and exploiting weaknesses in the MIFARE Classic card. It is common for these papers to conclude that the use of MIFARE Classic cards should be deprecated in favour of more secure cards. These papers seldom consider the practical implications of upgrading and make no suggestions for how an organisation can protect themselves whilst upgrading.

1.2 Applicability to Other Cards

Defence in depth is a security principle in which security measures are layered upon each other to provide redundancy when one security measure fails. Several of the countermeasures in this project can be used to enhance the security of other smart-cards and harden them against as yet unknown vulnerabilities.

1.3 Related Work

The security of MIFARE Classic cards has been the subject of much academic interest. The proprietary encryption algorithm CRYPTO1 was partially reverse engineered by Nohl and Plötz in 2007 [4] by depackaging* the chip and reverse engineering the gates as seen with a microscope. Garcia *et.al.* built upon this and fully reverse engineered the algorithm in 2008[1] by studying captured communication traces. Within the same paper, Garcia *et.al.* proposed the first practical attack. The card manufacturer, NXP, attempted (and failed) to get a court injunction to prevent the publication of the research.

Published research on digital signatures for NFC cards include a paper published on the NFC forum [5] and a paper by Markus Kilås [6].

1.4 Background

Over the past two decades, contactless smart-cards have become commonplace throughout the world. Amongst the most widely deployed are MIFARE Classic cards, with over one billion produced.

Communication between an NFC reader and a MIFARE Classic card is protected by a protocol designed to provide mutual authentication. The authentication protocol

*Slicing the chip open, exposing the gates.

uses a proprietary encryption algorithm called CRYPTO1. Security researchers have discovered weaknesses that allow an attacker to extract the secret keys and thus bypass authentication.

The goal of this project is not to completely "secure" MIFARE Classic cards; the security of the cards is fundamentally broken. The goal is to provide countermeasures to allow an organisation to better protect themselves during the several years it may take to upgrade a large deployment of cards. The countermeasures described in this project significantly increase the complexity and resources required to successfully carry out an attack.

1.5 MIFARE Classic Overview

The MIFARE Classic card is one of several NFC cards manufactured and sold by NXP semiconductors (formerly part of Philips). The card conforms to parts one to three of the four part ISO 14443 Type A standard. The fourth part of the standard describes the transmission protocol between a card and a reader. NXP deviated from the standard to include their own proprietary encrypted communication protocol.

The card is, in essence, a contactless memory card with read and write access protected by secret keys.

1.5.1 Logical Structure

The memory on the card is split into 16-byte blocks. These blocks are grouped into sectors, each sector having its own secret keys stored in its last block (the sector trailer). Blocks within a sector typically contain data that is logically connected as each block in the sector shares the same secret keys. For example, at the University of Cambridge one sector holds the CRSID* and student number whilst another sector holds data for door access to a particular college.

There are two variants of the card, 1K and 4K. The 1K and 4K cards differ in both the size of memory and how it is divided into sectors (see Figure 1.1 on the following page). The 1K card has 16 sectors, each of which has 4 blocks giving it a total memory of 1024 bytes. The 4K card has 40 sectors, the first 32 sectors have 4 blocks and the last 8 have 16 blocks as shown in Figure 1.1 on the next page, giving it a total memory of 4096 bytes.

When NXP introduced the 4K card it was desirable to have no more than 40 sectors†. The non-uniform distribution of blocks to sectors in the 4K card was likely motivated by a desire to retain backwards compatibility whilst keeping within the limit of 40

*The Common Registration Scheme IDentifier is the identifier issued by the University Information Services to identify students and staff.

†The card uses a lookup table that allows a reader to lookup which sector contains certain data. To address more than 40 sectors the lookup table would have to occupy an additional sector itself.

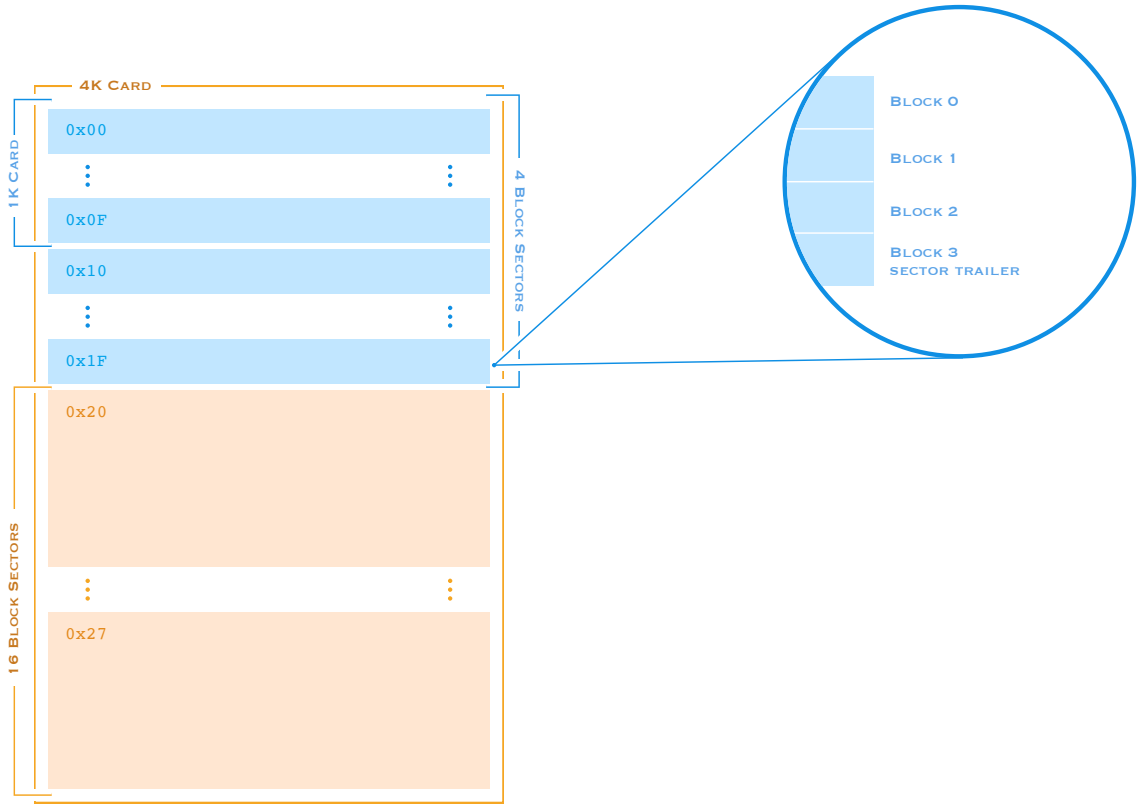


Figure 1.1: Logical Structure of Mifare Classic Cards

sectors. By keeping the first 16 sectors of the 4K card the same as the 1K card, organisations are able to start issuing 4K cards without upgrading all their existing readers.

1.5.2 Blocks

All blocks except the last one in each sector are data blocks and can be used for storing data. The last block in each sector is the sector trailer, it contains the secret keys and access bits. The configuration of the access bits determines which operations can be performed by each key and whether a data block is a read/write block or a value block. The data layout and available operations vary between the two types of data block.

Read/Write Block

A generic read/write block contains 16 bytes of arbitrary data. The data is stored without parity bits and thus if data integrity is vital then parity bits should be added to the data itself. The configuration of the access bits determine which keys can perform each operation.

Value Block

Value blocks contain a signed 4-byte value and a 1-byte pointer to another block. The 4-byte value is stored three times, twice non-inverted and once inverted as shown in Figure 1.2. The 1-byte pointer is stored both inverted and non-inverted twice.

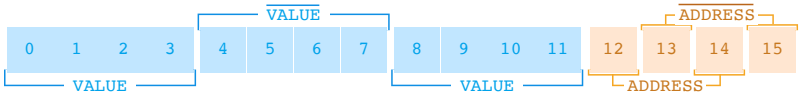


Figure 1.2: Value Block

Value blocks are used in electronic wallet applications such as public transport ticketing systems where data integrity is very important. Storing the value three times provides strong protection against data corruption. This level of redundancy is warranted when a single flipped bit could have a large financial impact. The 1-byte address is used to store a pointer to a block containing a backup of the data.

Sector Trailer

The last block of each sector is the sector trailer, it contains:

- **Access Bits**
  The access bits specify the operations that each key is allowed perform on each block of the sector.
- **Secret Keys**
  Secret key A is always present, secret key B is optional. If the access bits are such that the memory typically occupied by key B is readable by key A then those bytes are treated as data and cannot be used to authenticate. When the sector trailer is read any secret keys are masked with zeros.
- **User Data**
  The last byte of the access bits is unused, the MIFARE Classic specification explicitly states that it is available for user data rather than being reserved for future use[7].

The sector trailer is divided as shown in Figure 1.3 on the next page. The first six bytes contain key A. The next four contain the access bits, with the last byte (9) unused and available for user data. The remaining six bytes contain either key B or user data depending on whether the access bits allow it to be read.

1.5.3 Memory Operations

There are six memory operations that can be performed on a block.

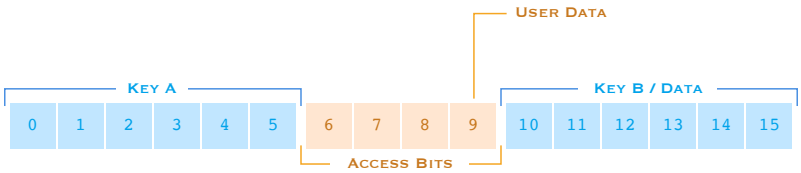


Figure 1.3: Sector Trailer

Read and Write

The read and write operations can be used wherever the access bits permit, irrespective of whether the block is a read/write block, value block or sector trailer. The operations read and write a block respectively. When reading the sector trailer zeros are returned in place of the keys.

Decrement, Increment, Restore and Transfer

These operations are only permitted on data blocks that are configured and formatted as value blocks. The decrement and increment operations decrement or increment a value block by a given argument and store the result in an internal register. The restore operation loads the value from a value block into the internal register[7]. The transfer operation transfers the value from the internal register into the value block[7].

Chapter 2

Preparation

This chapter begins by giving an overview of the most significant weaknesses in the MIFARE Classic card and describing the most effective attack. This knowledge is then utilised to devise a series of countermeasures that mitigate the risk of attack, including digital signature protection and a protocol for distributing revoked UIDs. At the end of the chapter, the viability of both using an existing MIFARE library, and writing my own is assessed.

2.1 Weaknesses in MIFARE Classic Cards

This section explains the four main weaknesses of the CRYPTO1 stream cipher and describes the impact they have on the overall security of the card.

2.1.1 Weak Pseudo Random Number Generator

Access to the memory on the card is protected by secret keys. The proprietary encryption protocol CRYPTO1 was designed to provide mutual authentication* between a card and reader.

Communication between the card and the reader starts with a three-pass challenge-response handshake as shown in Figure 2.1 on the following page. This handshake allows both the card and the reader to verify that the other knows the secret key without ever transmitting the secret key itself.

The card/tag nonce is generated using a hardware pseudorandom number generator. The PRNG consists of a 32-bit linear feedback shift register (LFSR). On each clock cycle the leftmost bit is discarded and a new bit shifted in by XORing bits 16, 18, 19 and 21 as shown in Figure 2.2.

The LFSR is 32 bits long and thus produces 32-bit nonces. However, as can be seen in Figure 2.2, the feedback function only uses the 16 rightmost bits, restricting

*With mutual authentication both entities authenticate each other.

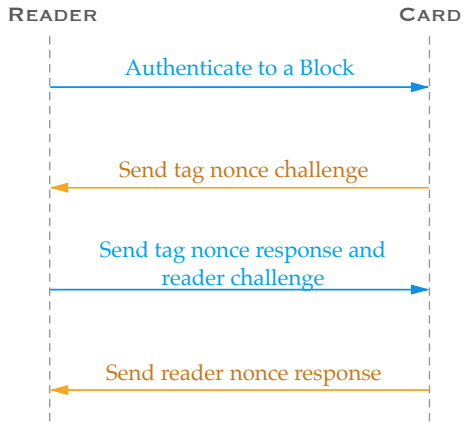


Figure 2.1: Three Pass Authentication Handshake

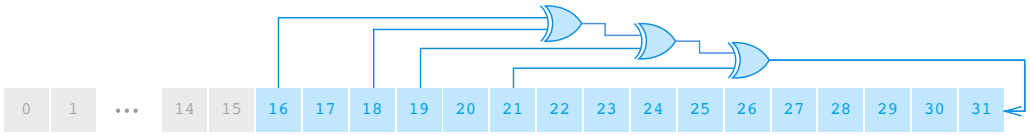


Figure 2.2: Pseudo Random Number Generator Schematic

the entropy of the generated numbers to 16 bits. Instead of being able to produce $2^{32} - 1$ $(4,294,967,295)$ different random numbers the PRNG can only produce $2^{16} - 1$ $(65,535)$.

The MIFARE Classic chip is powered by the electromagnetic field of the reader. The chip loses power when it is removed from the reader or if the field is turned off. When the chip loses power, the PRNG resets to a known state. An attacker can therefore accurately predict the "random" nonce values by resetting the card and then waiting a fixed number of clock cycles before requesting the nonce.

2.1.2 Parity Bits Leak Information

The ISO/IEC 14443-A standard requires that every byte of data transmitted be followed by a parity bit for transmission error detection. The parity bit specifies whether there is an even or odd number of "1" bits in the byte and thus allows detection of single bit transmission errors.

The standard is agnostic about the data being transmitted and makes no specific provision for authentication or encryption. To introduce encryption in a way that is compliant with the standard, the parity bits should be added after the transmission data is encrypted. The MIFARE Classic not only adds the parity bits prior to encryption but also encrypts them with the same bit of the keystream as is used for the next bit of the plaintext as shown in Figure 2.3. In general, this reduces the entropy of every byte transmitted by one bit as it leaks whether the first bit in a byte is the same as the parity bit of the previous byte.

During the authentication handshake shown in Figure 2.1, when the reader responds

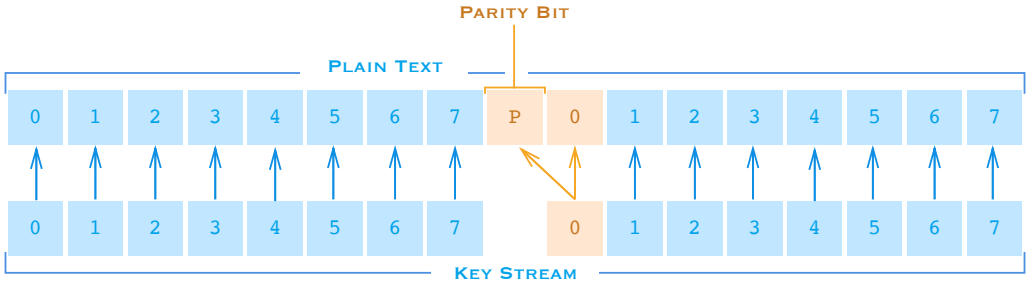


Figure 2.3: Key Stream Bit Reuse on Parity Bits

to the card nonce, the card verifies the parity bits before checking whether the response is valid. If the parity bits are incorrect, the card ceases communication. If all eight parity bits are correct, the card responds with a 4-bit error code. There are 8 parity bits and a randomly chosen parity bit has a 50% chance of being correct. Therefore when choosing a response randomly, all 8 parity bits are correct with probability 1 in 256. The error code is encrypted despite the authentication failing, thus allowing the recovery of four bits of the keystream by XORing the received encrypted error with the known plaintext value.

2.1.3 LFSR Can Be Rolled Back

Communication between the card and the reader is encrypted with the proprietary encryption algorithm CRYPTO1. A successful authentication handshake initialises the 48-bit CRYPTO1 linear feedback shift register (LFSR) in both the card and the reader. Note that this is a different LFSR than the one in the PRNG.

CRYPTO1 is a stream cipher; at each clock cycle a keystream bit is generated from the LFSR using the filter function as shown in Figure 2.4. The filter function is constructed using 3 smaller filter functions (f_a, f_b, f_c), each of which returns either a 1 or a 0 depending on the values of its inputs.

Every clock cycle the LFSR shifts a bit to the left, discarding the leftmost bit and inserting a new bit from the feedback function by XORing several bits together as shown. The keystream bit is XORed with the next plaintext bit before transmission. Upon receipt of an encrypted transmission, the reader XORs each bit of the ciphertext with successive bits of the keystream to recover the plaintext.

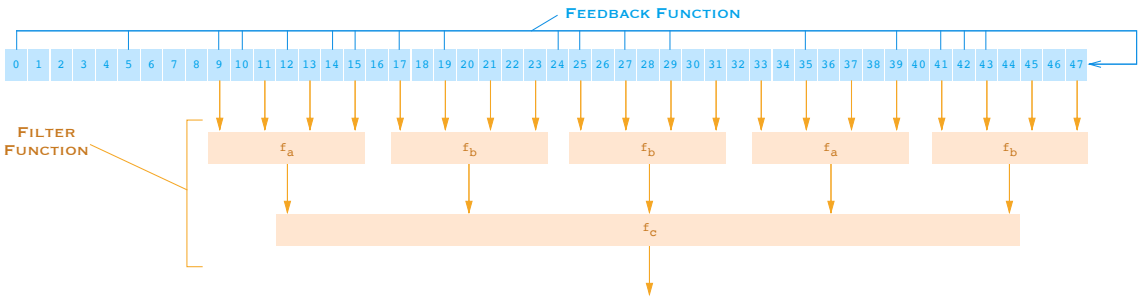


Figure 2.4: CRYPTO1 LFSR Construction

If the LFSR state is known then it can be rolled back, recovering both its initial state and all keystream bits. To roll back the LFSR state by one clock cycle, each bit is shifted one to the right. The value of the new bit (position 0) can be determined by using the bit that we shifted out of position 47. This bit was produced by XORing 18 bits together, 17 of which we know and the 18th is the new bit.

2.1.4 LFSR State can be Recovered

As can be seen from Figure 2.4 on the previous page, all the 20 input bits to the filter function are odd bits from the LFSR. This allows an attacker that has recovered part of the keystream to invert the filter function and recover the LFSR state by doing the following:

1. Consider the first bit of the recovered keystream ks_0 and the LFSR bits that input into the filter function (bits 9, 11, ..., 47). Compute all possible filter inputs that produce ks_0. The structure of the filter function guarantees that there will be exactly 2^{19} such possibilities.
2. For each of the possibilities:
   (a) Shift the LFSR 2 bits to the left and consider the keystream bit 2 positions after the previous one. In other words, we iterate over either the even or odd keystream bits.

   The filter function still has 19 of the previous 20 bits as inputs and a new bit from the feedback function.

   (b) The new bit could be either a 1 or a 0, check if each of these possibilities result in the correct keystream bit when put through the filter function.

   - If neither result in the correct keystream bit, then this sequence of LFSR bits has been eliminated, so we remove it.
   - If one results in the correct keystream bit, then this sequence of LFSR bits is extended with this new bit value.
   - If both result in the correct keystream bit, then we duplicate the sequence and extend one sequence with a 0 and one with a 1.

   (c) We repeat this until we have run out of keystream bits. We now have a list of sequences of alternate LFSR bits, each of which produces the alternate bits of the keystream.
3. Having recovered the LFSR sequences corresponding to the even bits of the keystream we now repeat this, starting with ks_1 to recover the sequences for the odd bits of the keystream.
4. Combining the odd and even sequences provides candidate sequences for the entire LFSR state. To do this, the bits of the sequences are checked to see if they satisfy invariants that should hold due to the construction of the feedback function.

The number of candidate states returned varies depending on how many keystream bits were used. When 32 bits are used, on average 2^{16} candidate states are returned. When 64 bits are used only one state is returned.

2.2 Attack

The weaknesses outlined in the previous section can be exploited via a range of attacks. This section describes one of the most effective attacks. The attack only requires access to a card and doesn't need specialist equipment.

2.2.1 Secret Key Recovery from Nested Authentication

When a reader with an existing authenticated session wants to re-authenticate to use a different secret key the protocol differs slightly and the three pass handshake shown in Figure 2.1 on page 15 is encrypted. Therefore, if we are able to predict the plaintext (tag nonce) it is possible to recover 64 bits of keystream by XORing the plaintext with the ciphertext.

The first sector on MIFARE cards is readable with a default key and thus it is always possible to obtain an authenticated session with which to mount this attack.

Due to the PRNG weakness, the nonces have at most 16 bits of entropy. The parity weakness further reduces this to just 6 bits and thus just 64 candidate nonces. Enumerating each possibility is feasible but in practice we can do better, as even the timing information from off-the-shelf readers is sufficient to determine roughly what state the PRNG is in and thus which of the 64 candidate nonces was most likely used.

As described in Section 2.1.4 on the previous page, the keystream bits can then be used to derive the LFSR state, which can be rolled back to its initial state, recovering the secret key.

2.2.2 Brute-Force Key Search

Prior to the reverse engineering of CRYPTO1 a brute-force attack had to be performed against the card directly. This required trying all possible 48-bit keys against the MIFARE Classic card. The MIFARE Classic specification [7] states that the communication required to try a key should take a minimum of 5 ms, it would therefore take in excess of 40,000 years to perform an exhaustive search on the entire keyspace.

Now that the encryption algorithm has been reverse engineered, offline brute-force attempts can be carried out against captured communication with a card. The time required in an offline brute-force is inversely proportional to the cost. In 2009 a $10,000 computer could crack DES (a much slower and more gate hungry algorithm)

in a week [8]. A 48-bit key is therefore insufficient to provide robust security with the computing resources available today, even with a better cipher.

2.3 Attack Countermeasures

In this section I explore various countermeasures to the above attacks, explaining both how they work and how they mitigate the risk of attack.

2.3.1 Digital Signature Protection

The security of the MIFARE Classic card is compromised so severely that any data stored on the card no longer has data integrity or confidentiality. If the data corresponds to a balance then the weaknesses allow an attacker to commit fraud. Within the context of the University of Cambridge, an attacker could change the student identifier or CRSID to gain unauthorised access to buildings.

Digital signatures provide cryptographic assurance of the integrity of the data, irrespective of the security of the card itself. Data signed with a robust digital signature algorithm cannot be created or modified without access to the signing key, as doing so would lead to an invalid signature.

Protection Against Cloning

Digital signatures do not provide any inherent protection against cloning. The presence of a valid signature provides nonrepudiation — strong cryptographic assurance that the data was created by the signer. However, an adversary with read access can copy both the data and the signature to clone the card.

MIFARE Classic cards have a read only unique identifier (UID) that is set during manufacture. By also signing the UID an attacker won't be able to clone the data onto another MIFARE Classic Card as the signature will be invalid.

An attacker can bypass this, however, if they are able to set the UID. Specialist hardware like the Proxmark 3 can emulate a MIFARE Classic card with an arbitrary UID.

Requirements

The primary purpose of this dissertation is to explore countermeasures that organisations with existing MIFARE Classic deployments can implement until they are able to migrate to a more secure card. Given this, it is desirable that any Digital Signature proposal meets the following criteria.

- **Backwards Compatible**
  In the unlikely event that is was feasible to update all readers and cards in one go then the organisation could migrate to a more secure card instantly.

  In the more realistic scenario where readers and cards will be updated to incorporate digital signatures gradually, it is vital that the scheme is backwards compatible. Readers that have not been updated should not be affected and should continue to operate (without enhanced security).

- **Agnostic of Card Structure**
  Different organisations structure the data on their cards in different ways. Even a single organisation may not have a structure that is consistent across all cards. The digital signature implementation should be flexible enough to work with all typical deployments rather than being specific to any one card layout.

- **Allow Key Updates**
  Whilst every effort should be made to prevent the private keys from being leaked, it would be foolish to not plan for this eventuality. Following a breach the private keys should be changed, and the system should be able to deal with this in a way that minimises disruption.

- **Support Multiple Signatures**
  A single card may contain data from different sub-organisations, each of which may want to add a signature. For example, within the University of Cambridge, central university may wish to sign some data such as the student ID and expiry date. This should not preclude colleges from signing their own data independently.

2.3.2 Key Diversification

Without key diversification, every card within a deployment has the same set of secret keys. Should an attacker be able to extract the secret keys from one card they will be able to use them to access any other card within the deployment. With key diversification, the secret keys for a particular card are cryptographically derived from a master key using the unique ID of the card.

With key diversification an attacker is forced to extract the secret keys from any card they wish to clone. Instead of requiring a few seconds of communication with the target card the attacker may require several minutes to extract all 80 keys of a 4K card. It would be harder for an attacker to sustain this prolonged access covertly.

2.3.3 Interaction Counter

Key diversification makes it more challenging but not impossible for an attacker to dump the data from a card. Digital signature protection makes it harder for an attacker to use dumped data to create a clone as the UID of the card is signed.

Neither of these, however, make it impossible. It is thus desirable to be able to detect the presence of cloned cards.

By using an interaction counter it is possible to detect the use of some cloned cards. The counter consists of an integer and occupies one 16-byte block. When the card is placed on a reader, the counter is incremented. Networked readers report the current value to a central system. The central system checks that any counter value reported is more than the previously highest reported value for that card. When a card is cloned, if either the original or the clone is used, the counter in each gets out of sync and thus when the other card is used the counter value will be less than the previously reported value, indicating that the card is cloned.

Whilst this method won't detect all cloned cards, the possibility of detection would act as a deterrent.

Limitations

This does not protect against the situation where the owner of the cloned card is complicit in its cloning. If this is the case then the counters can be kept in sync.

When a reader detects the presence of a clone it doesn't know whether the current card is the clone or the cloned card. It may be undesirable therefore to reject access.

Only networked readers can report cloned cards, however non-networked ones can still increment the counter, allowing detection by subsequent access to an online reader.

2.3.4 Emulated Card Detection

With the proposed digital signature protection, an attacker not only has to clone the contents of the card but also the UID. The UID in MIFARE Classic cards cannot be changed. An attacker therefore has to either obtain a knock-off card that allows changing the UID or emulate the card using a device such as the Proxmark 3.

Recall that MIFARE Classic cards have a hardware based random number generator (PRNG) used to generated nonces. Also recall that the PRNG is flawed as the feedback function only takes input from the rightmost 16 bits as shown in Figure 2.5 resulting in the generated 32-bit numbers having at most 16 bits of entropy.

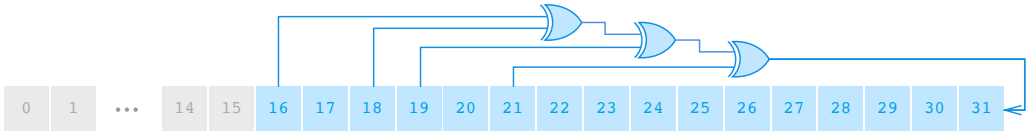


Figure 2.5: Pseudo Random Number Generator Schematic

Most emulators do not implement a firmware/software version of the MIFARE Classic's PRNG and instead use their own source of random nonces. The random nonces

generated by the PRNG are used in the initial authentication handshake between the card and the reader as shown in Figure 2.1 on page 15.

Since the hardware PRNG can only generate $2^{16} - 1$ (65,535) different nonces it is easy to check whether a given nonce could have been generated by the PRNG. This allows the reader to accurately identify emulated cards that don't use the same PRNG. The reader can then reject access and flag for review.

Limitations

Assuming the nonces generated by the emulated card are uniformly picked at random from all $2^{32} - 1$ possibilities, there is a one in 65,535 chance that it will be in the PRNG range and thus be indistinguishable, using this method, from a genuine MIFARE Classic Card.

Given the weakness in the PRNG is a hardware implementation flaw rather than a weakness in the authentication protocol it is possible that NXP could fix the flaw in the PRNG in future cards in a backwards compatible way. If the readers were programmed to reject these cards then this would cause disruption.

I am of the opinion that it is unlikely that NXP will alter the implementation. The cards have been produced for eight years since the weaknesses were published and NXP have yet to make a change. The MIFARE Classic protocol is fundamentally broken and whilst fixing the PRNG would mitigate some of the attacks, it would not prevent all of them.

2.4 Card Revocation

Cards are often issued with an expiry date after which they can no longer be used. Without digital signature protection, an adversary can use the attacks previously described to change this expiry date, potentially allowing them to gain unauthorised access. By including the expiry date in the data to be signed, an attacker cannot use an expired card simply by changing the expiry date.

There are a variety of circumstances under which it is necessary (or at least highly desirable) to revoke a card before its originally planned expiry date. Within the context of the University of Cambridge, reasons for revoking a card early include:

- The card is lost, stolen or broken
- A student or member of staff leaves early

In this section I outline some existing solutions, explain their shortcomings when applied to MIFARE Classic deployments and propose a card revocation system that uses the cards to distribute a list of revoked cards from online (networked) readers to offline (non-networked) readers.

2.4.1 Existing Solutions

Revoke Card in Central Database

Networked or "online" card readers query a central database each time a card is presented. Revoking access via these readers is thus simply a case of updating this central database to either remove the card or mark it as revoked.

There are a variety of factors that may make the installation of a networked reader impractical. Within the University of Cambridge, offline door-access readers (those without a network connection) are common. Currently, to revoke access to these readers, they have to be manually updated. The resources required to update each offline reader when a card is lost are sufficiently high that it is seldom done, resulting in lost or stolen cards remaining active.

Hotel Key-Card Solution

Some hotels issue key-cards to access rooms. The door to the room has an integrated, unnetworked reader. The hotel solution relies on each card only having access to a single reader and cannot easily be extended to an arbitrary network of readers and cards.

If a guest checks out or loses their room key then the hotel issues a new key and revokes the old one. This usually works by incrementing a per-room counter stored on the card each time a new one is issued. The reader maintains its own counter that it updates whenever a card with a higher counter is used. The reader reject cards with a counter lower than their internal one. Therefore the act of using a new card automatically revokes previously issued ones.

Within the context of a hotel this scheme has several benefits:

- **No network or power required**
  It would be expensive to run a network connection to each door in the hotel so there are significant cost savings by removing the need for one. The readers only require a very small amount of power and can usually run for years from a small integrated battery.
- **Guests can extend their stay**
  If a guest extends their stay their card will continue to work.
- **Multiple cards can be issued**
  Multiple cards can be issued to guests by issuing cards with the same counter value. If a second guest staying in the room arrives later then a second card can be issued with the same counter value without reissuing the previous cards.

2.5 Card Revocation Gossip Protocol

Whilst offline readers aren't networked in a traditional sense, they are part of a network of readers through which cards travel. Cards can be used as a medium to transfer data between readers and thus can be used to distribute a list of revoked cards from online readers to offline ones.

2.5.1 Requirements

The card revocation protocol is for use in deployments with a mixture of networked and non-networked readers, such as the University of Cambridge. The protocol distributes the revocation list to offline readers, using the cards as a medium for communication. The network is unreliable and unpredictable and the protocol should be tolerant of this, making specific provisions for:

- varying numbers of online and offline readers
- different network topologies
- varying numbers of cards
- varying numbers of revoked cards

2.5.2 Development Model

Given the importance of the protocol performance, and the unpredictable nature of large scale simulations, an iterative approach to development is appropriate. This allows changes to the protocol that are motivated by particular areas of poor performance.

2.5.3 Simulation Suite

A suite that simulates a network of readers and cards in varying configurations is necessary to assess the performance of each implementation. It is of significant benefit to use the simulation suite during development to highlight particular situations under which the implementation performs poorly and inform changes.

2.6 MIFARE Libraries

In order to implement the digital signatures and card revocation gossip protocol it is necessary to be able to communicate with a MIFARE Classic card. After investigating the viability of using an existing MIFARE library I decided that writing my own would be preferable for reasons outlined below.

Both Windows and OS X have built in "smart-card" APIs for interacting with both contactless and contact smart-cards. Using either of these APIs would make my implementation platform-dependent, something I wanted to avoid. Both APIs are also reasonably high-level and don't expose sufficient low-level functionality to implement all of my proposed countermeasures.

Libnfc is an open source C library with support for a variety of NFC cards and readers. Libnfc is cross-platform and exposes enough of the low-level functionality of the cards to implement all of my proposed protections. It would have been possible to implement my project entirely in C using libnfc. The decision to write my own library in the Go programming language has several benefits and was motivated by a variety of factors:

- **Go is a high level language**
  Go is an object orientated, garbage collected, high-level language, which makes writing and maintaining certain aspects of the implementation easier.
- **Ability to write unit tests**
  It is a well established software-engineering practice to write unit tests for code. Had I written the software directly on top of libnfc, it would have been prohibitively difficult to write tests that didn't depend on a physical card. By writing my own library I'm able to provide sufficient abstraction to allow mock versions of hardware readers and cards for tests.
- **Ability to write simulations to benchmark**
  It would be impossible to write the simulation suite on top of libnfc. By writing my own library I can write code that works with both simulated readers and cards as well as physical ones.
- **Suitability of Go**
  Go was chosen as the language to implement the core library in for a variety of reasons including:
  - The project makes extensive use of cryptographic operations and Go has excellent support for these in its standard library
  - The asynchronous nature of NFC communication is a natural fit for Go's concurrency primitives such as goroutines and channels.
  - Go can be cross compiled to produce statically linked binaries for all major platforms.
- **Would provide a significant contribution to the community**
  Many of the potential problems I faced with using libnfc directly also affect others developing software for MIFARE Classic cards. By releasing an open source library I will have made a substantial contribution to the community.

Chapter 3

Implementation

In this chapter, I explain how I implemented my project, justifying decisions made and describing the difficulties I faced. Explanations of MIFARE concepts not previously introduced are included where appropriate. The implementation consists of the following:

- A library for interacting with, and simulating MIFARE cards
- A library for reading and writing signed data to MIFARE cards
- A simulation suite for simulating large networks of readers and cards
- A gossip protocol for distributing revoked UIDs to offline readers

3.1 MIFARE Library Implementation

The MIFARE library is a library written in Go, upon which the rest of my implementation sits. It provides an API for interacting with MIFARE cards and a software implementation of the card to support unit tests and simulations.

3.1.1 Structure of Library

The library is split into two parts, interfaces and drivers*. The interfaces specify the publicly exposed objects the library provides and their public method signatures. The drivers are specific implementations that satisfy these interfaces.

By splitting the library in this way, the public API exposed by the library is abstracted from the implementation. This means that any code built on top of the interfaces exposed by the library will work with any driver (implementation) that satisfies the interfaces. This flexibility has many benefits, including:

*I didn't use the more suitable object-oriented term "implementations" to prevent it from becoming overloaded within this chapter.

- **Support mock drivers**
  Code that uses the library can be easily tested using a mock driver. The mock driver provides an in-memory implementation of readers and cards that satisfy the exported interfaces. Without this, any unit test would be dependent on physical hardware — making it more fragile and harder to run regularly.
- **Support platforms without libnfc**
  The library contains a driver (implementation) that uses libnfc and a mock one as this was required for my project. It would, however, be possible to write a driver that does not depend on libnfc. This would be useful on platforms such as Android where a system API is provided and libnfc may not be supported due to security sandboxing restrictions.

3.1.2 Exported Interfaces and Types

The public API consists of a series of interfaces and types. In this section I explain them by including their definitions and providing background explanation where appropriate.

Driver and Device

The `Driver` is the base interface that has a method for listing available NFC readers and connecting to them.

```
type Driver interface {
    // Name returns a string name for this driver e.g "libfreefare wrapper". It is
    // typically only used in debug output.
    Name() string

    // ListDevices returns a list of connection strings or an error.
    ListDevices() ([]string, error)

    // Open takes a connection string and returns the Device corresponding to it
    // or an error. If conn is empty then it connects to the first available device.
    Open(conn string) (Device, error)
}
```

A `Device` represents a connected NFC reader. It supports listing tags (cards) within the reader's field and closing the connection to the reader.

```
// Device represents an NFC device (e.g. USB reader)
type Device interface {
    // ListTags returns a list of Tags (cards) that are currently within the
    // reader's field
    ListTags() ([]Tag, error)

    // Close closes the connection to the device
    Close() error
}
```

Tag and TagType

The `Tag` interface represents an arbitrary NFC tag. By separating the `Tag` and `ClassicTag` interfaces it makes it possible to add support for other types of card in the future without breaking backwards compatibility.

```
// Tag represents an NFC tag
type Tag interface {
    // UID returns the card's unique ID
    UID() string
    Type() TagType
}

// TagType represents the type of RFID tag (only Mifare Classic tags are
// supported)
type TagType int

// The types returned by tag.Type()
const (
    TagTypeUnknown TagType = iota
    TagTypeClassic1K
    TagTypeClassic4K
)
```

ClassicTag

The `ClassicTag` interface embeds the `Tag` interface, this is analogous to extending in a more traditional object orientated language. The `ClassicTag` interface contains methods for each of the six operations that are available on a MIFARE Classic card, as well as methods for interacting with the MIFARE Application Directory (MAD). Both the MAD and the types referenced in the method signatures are explained in the following sections.

```
// ClassicTag represents a Mifare Classic tag
type ClassicTag interface {
    Tag

    // ReadMAD returns the Mifare Application Directory for the card
    // or an error
    ReadMAD() (MAD, error)

    // WriteMAD writes the MAD to the tag using the provided key
    WriteMAD(m MAD, sector00KeyB, sector10KeyB Key) error

    // ReadBlock reads the specified block using the specified key
    ReadBlock(block byte, key Key, keyType KeyType) ([16]byte, error)

    // WriteBlock writes the specified block using the specified key
    WriteBlock(block byte, data [16]byte, key Key, keyType KeyType) error

    // DecrementBlock decrements the value in the given block by the given amount
    // and stores the result in the internal register
    DecrementBlock(block byte, amount uint32, key Key, keyType KeyType) error

    // IncrementBlock increments the value in the given block by the given amount
    // and stores the result in the internal register
    IncrementBlock(block byte, amount uint32, key Key, keyType KeyType) error

    // RestoreBlock copies the value from the given block into the internal register
    RestoreBlock(block byte, key Key, keyType KeyType) error

    // TransferBlock writes the value from the internal register into the given block
    TransferBlock(block byte, key Key, keyType KeyType) error
```

```
    // ReadApplication looks up the application ID in the Mifare application
    // directory and then reads the corrresponding sectors.
    // It returns the number of bytes read or -1 and an error
    ReadApplication(m MAD, aid AID, key Key, keyType KeyType) ([]byte, error)

    // WriteApplication looks up the application ID in the Mifare application
    // directory and then writes the data to the corrresponding sectors.
    // It returns the number of bytes written or -1 and an error
    WriteApplication(m MAD, aid AID, data []byte, key Key, keyType KeyType) error
}
```

MIFARE Application Directory

The MIFARE Application Directory (MAD) is a lookup table that stores an Application Identifier (AID) for each sector on the card. By using sector pointers in the MAD instead of storing data in fixed sectors, the memory on the card can be used more efficiently.

By considering the University of Cambridge it is clear why this is useful. Every college and department may need to put data on the card. There are insufficient sectors to assign a static sector to each of these. However, a member of the University will only have associations with a handful of these sub-organisations and thus the memory available is sufficient when using the Application Directory.

Use of the Application Directory is also useful for anti-collision. A user may have multiple MIFARE Classic cards in their wallet, for example an old Oyster card and a University Card. The reader can quickly determine which of the cards is the correct one by searching the Application Directory for the presence of their Application ID.

MAD Structure

On a 1K card, the MAD is stored in sector `0x00`, on a 4K card the MAD is stored across sectors `0x00` and `0x10`. The MAD specification dictates that the MAD should have the fixed read key (key A) of `A0A1A2A3A4A5` and a private read/write key.

The first block in sector `0x00` contains the unique identifier of the card and manufacturer data. This block is set when the card is manufactured and cannot be written to. The remaining two data blocks of sector `0x00` and the three data blocks of sector `0x10` contain the application directory as shown in Figure 3.1 on the following page.

The `MAD` interface has a method for looking up the AID for a sector and for allocating sectors for a new application of a given size.

```
// MAD represents the Mifare Application directory. On 1kB cards
// the MAD is located in sector 0x00, on 4kB cards it is located in both
// sectors 0x00 and 0x10.
//
// The MAD is essentially a map from sector numbers to the corrresponding
// application ID.
type MAD interface {
    //AIDForSector returns the AID for the corresponding sector
    AIDForSector(sector byte) AID

    // AllocApplication allocates enough sectors to store size bytes and then returns
```

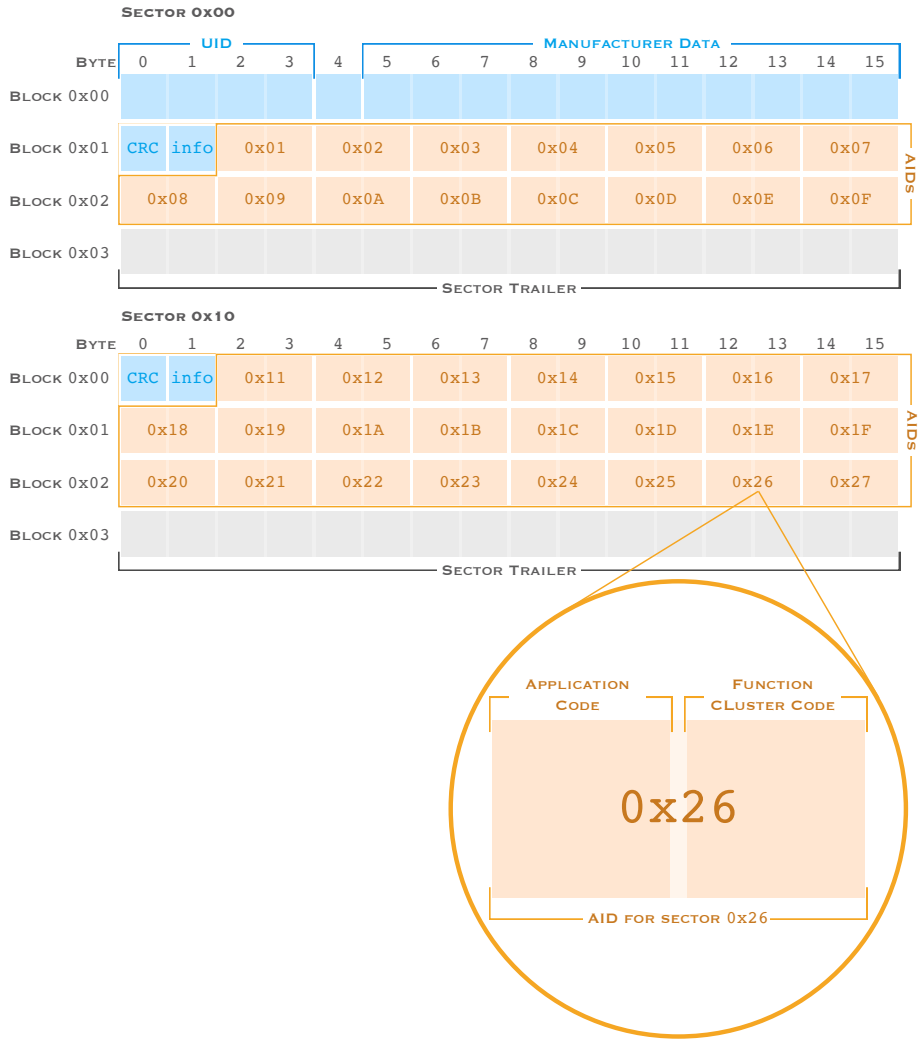


Figure 3.1: Structure of the MIFARE Application Directory

```
    // the allocated sectors if successful. If the application has already been
        allocated
    // nil is returned.
    AllocApplication(aid AID, size uint) ([]byte, error)

  // FindApplication returns a slice of sectors that contain the specified application
  FindApplication(aid AID) []byte
}
```

Application Identifiers

Application IDs are globally registered with NXP, thus preventing two organisations from using the same application ID. The AID consists of two bytes, a one-byte function cluster code and a one-byte application code. The function cluster code determines the classification of application, some examples are:

- **Card Administration** (0x00)
- **Airlines** (0x08)

- **Academic Services** (`0x58`)
- **Ministry of Defence, Netherlands** (`0x4A`)
- **Ski Ticketing** (`0x50`)

The application code is incremented each time a new AID is registered within a function cluster code.

The library exports the following interface for an AID:

```
// AID represents an Application ID.
// AIDs consist of a function cluster code and an application code. The former
// identifies the broad category of the application (e.g. bus services / access
// control and security) and the latter the specific registration.
// AIDs must be registered with NXP, a list of current registrations can be found
// here: http://www.nxp.com/documents/other/MAD_list_of_registrations.pdf
type AID interface {
    ApplicationCode() byte
    FunctionClusterCode() byte
}
```

Given the simplicity of the AID, the library provides a universal implementation that each driver can use. The implementation is simply a 2-byte array with methods for returning the function cluster code and application code.

The library has constants for each function cluster code. This makes code that uses the library easier to read as rather than an opaque byte value a descriptive constant name is used.

The MIFARE specification defines some reserved AIDs for card administration. These AIDs are also provided.

```
var (
    // FreeAID is used for sectors that are free
    FreeAID = NewAID(CardAdministration, 0x00)

    // DefectAID is used when the sector is defective, e.g. secret keys are
    // destroyed or unknown
    DefectAID = NewAID(CardAdministration, 0x01)

    // ReservedAID is used when the sector is currently free but has been reserved
    ReservedAID = NewAID(CardAdministration, 0x02)

    // AdditionalDirectoryInfoAID is useful for future cards where the directory
    // needs more sectors
    AdditionalDirectoryInfoAID = NewAID(CardAdministration, 0x03)

    // CardHolderInfoAID is used for a sector that contains card holder info in
    // ASCII format
    CardHolderInfoAID = NewAID(CardAdministration, 0x04)

    // NotApplicableAID is used when this sector doesn't exist on the card
    NotApplicableAID = NewAID(CardAdministration, 0x05)
)
```

3.1.3 Drivers

The library contains two drivers, one which sits on top of `libnfc` and interacts with physical cards and a mock driver which is used for testing and simulation.

Because of the way the library is structured a third party could implement their own driver, provided it satisfies the above interfaces.

Libnfc Driver

The libnfc driver doesn't interact with `libnfc` directly. As shown in Figure 3.2, the driver sits on top of a "thin binding"* that allows it to call C functions from within Go. This binding sits on top of `libfreefare`, the part of `libnfc` which has support for MIFARE cards.

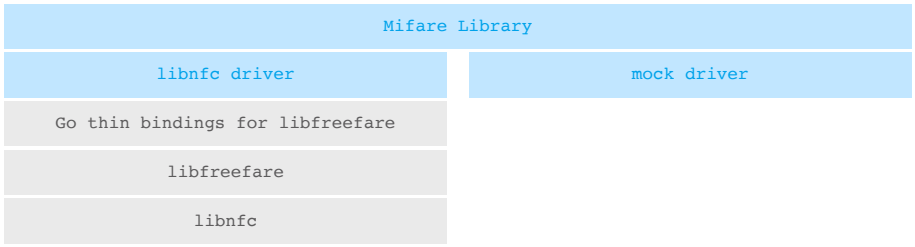


Figure 3.2: Structure of the MIFARE Library

Mock Driver

The mock driver provides a software implementation of the MIFARE Classic card. This is useful for both testing and simulation. The scope of the mock driver is sufficiently larger than that of the `libnfc` driver due to the necessity of implementing all of the card logic in software.

Structure of Mock Card

As shown below, the mock card is a struct containing a field specifying whether it is a 1K or a 4K card and an array of blocks.

```
type classicTagImpl struct {
    tagType mifare.TagType
    blocks  [256][16]byte
}
```

Access Bits

The access bits in the sector trailer control which memory operations are permitted by each key. There are 12 access bits, each access bit is stored once non-inverted and once inverted as shown in figure 3.3 on the following page. Whenever an operation is performed, the format of the access bits is verified and if a violation is detected the sector is irreversibly blocked, thus preventing corruption of a single bit from allowing unintended access.

The access bits are split into four sets of three bits (in figure 3.3 on the next page $C2_0$ denotes the second bit of the first set). Each block in the four-block sectors

*A thin binding allows code in one language to call functions from a library written in another. A thick binding provides its own API that sits on top of the library, making the exposed API more idiomatic to the new language.

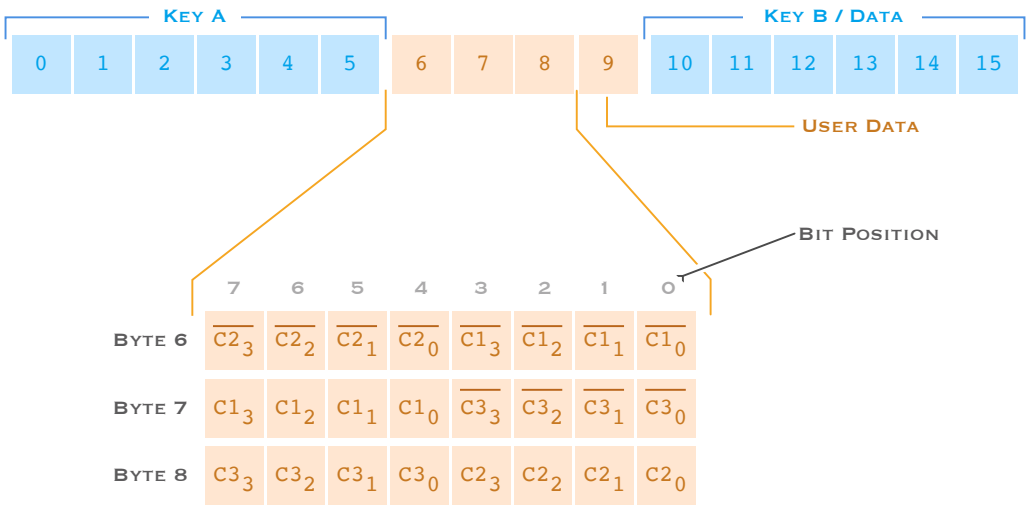


Figure 3.3: Access Bits Layout

including the sector trailer has its own set of three access bits and thus each block can have different access conditions. For 16-block sectors the 4 sets of access bits are divided between the blocks such that one set of access bits ($C1_3C2_3C3_3$) still controls access to the sector trailer and each of the remaining three sets of access bits controls access to five contiguous blocks as shown in figure 3.4.

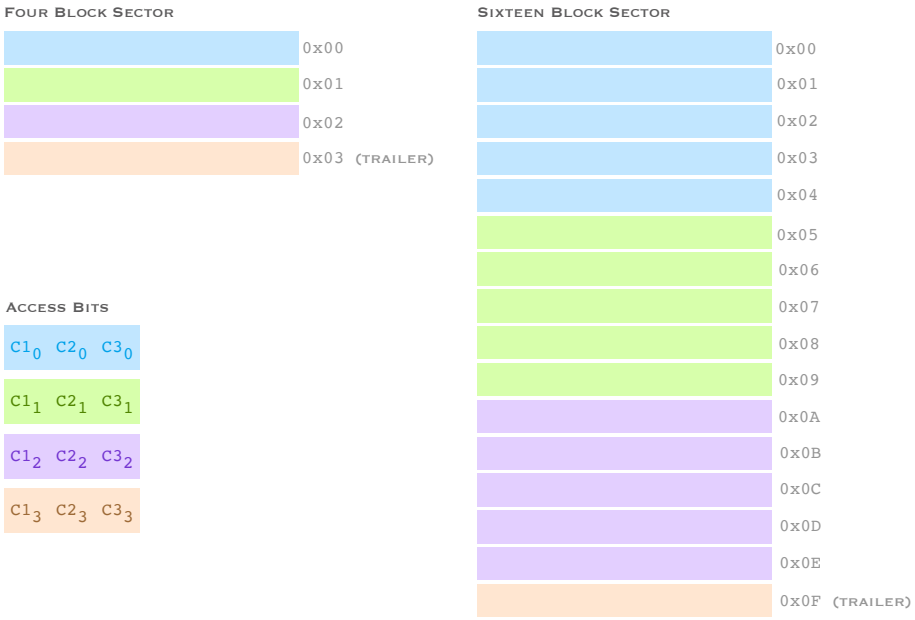


Figure 3.4: Block Division for Access Conditions

3.2 Digital Signature Protection

Recall from Section 3.1.2 on page 29 that the MIFARE Application Directory is a lookup table stored in sectors `0x00` and `0x10`. For each sector on the card, the

MAD stores the application ID (AID) corresponding to the data in that sector. By registering an AID for digital signatures and using the MAD the signature can be placed anywhere on the card where there is sufficient available space. By not relying on a static sector number the scheme can remain agnostic of the existing data layout.

Signature Structure

The signature consists of a 16-byte header followed by the raw signature bytes. The header contains the following as shown in figure 3.5 on the following page.

- **Signature Scheme Version** (1 byte in sector trailer)
  To allow for backwards incompatible changes to the scheme a 1 byte version number is included. This is set to `0x01` for this version of the scheme. Without this, any backwards incompatible changes would require a new signature AID. The byte is stored in the sector trailer.
- **Key Identifier (KID)** (2 bytes)
  The KID is an opaque identifier that the organisation increments every time they use a different key or signing algorithm. This KID allows a reader to be able to determine which key and signing algorithm was used to create a signature and therefore provides support for changing the signing key or algorithm.
- **AIDs** (14 bytes)
  The last 14 bytes of the header specifies the data that is signed, it consists of a sequence of 2 byte values. Each 2 byte value is the AID of an application that is signed. Within this context the "not applicable" AID (`0x0005`) is repurposed to mean the card UID.

The structure and length of the signature bytes section depends on the signing algorithm used. The scheme is agnostic of this, the KID specifies which key and signing algorithm was used. This allows third parties to implement their own signing algorithms if they are not satisfied by the implementation provided.

Signature Scheme

The signature scheme specifies which data and in what order it is passed to the signing algorithm. The data to be signed consists of the following:

1. **The signature header**
2. **7 bytes for the application lengths**
   For each AID in the signature header, a byte is included specifying the number of blocks of data for that application. If the AID is `0x0000` then `0x00` is the byte value used.
3. **The application bytes**
   For each AID in the signature header, the application data is added. If the

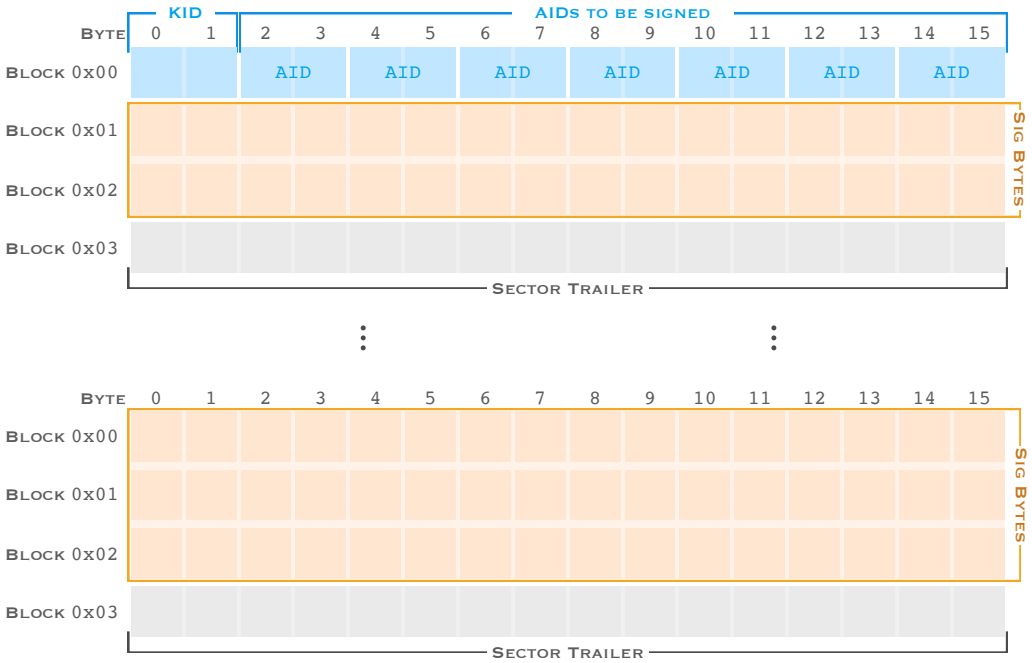


Figure 3.5: Signature Structure

value corresponds to the "not applicable" AID then the UID is added. If the AID is `0x0000` then no data is added.

Pseudo code of the signature algorithm is included below.

```
payload = []
payload.append(header.kid)

for aid in header.aids:
    payload.append(aid)

    if aid == 0x0000:
        payload.append(0)
    else if aid == NotApplicableAID:
        payload.append(len(uid))
        payload.append(uid)
    else:
        payload.append(num_blocks(aid))
        payload.append(get_data(aid))

def get_length(aid):
    if aid == 0x0000:
        return 0
    if aid == NotApplicableAID:
        return len(uid)
```

3.2.1 Digital Signature Library

The digital signature library is written in Go and sits on top of my MIFARE library. The implementation provides the following:

- **Elliptic Curve Keypair Generation Program**
  The library contains a method and command line interface for creating new

elliptic curve key-pairs.

- **Method for Allocating Signature**
  This method is used to provision a new signature. It does the following:
  - Allocate sectors for the signature by writing to the application directory.
  - Update the sector trailers with the new keys.
  - Write the signature header.
- **Method for Writing Signed Data**
  This method wraps the `WriteApplication` method from the MIFARE Library and additionally updates the signature.
- **Method for Reading Signed Data**
  This method wraps the `ReadApplication` method from the MIFARE Library and checks that the signature is present and valid.

Signing Algorithm

As previously mentioned, the scheme does not dictate which signing algorithm to use. However, for the library to be plug-and-play it would have to contain at least one signing algorithm implementation. I chose to implement the elliptic curve digital signature algorithm (ECDSA) as it produces much smaller signatures for the same level of security as its RSA equivalent. Keeping the memory footprint of the signatures as small as possible is desirable given the memory constraints of the card.

Elliptic Curve

An elliptic curve has to be chosen before creating an elliptic curve private key. The National Institute of Standards and Technology has standardised several curves including `P-224`, `P-256` and `P-384`. Each curve has slightly different properties.

There is a suspicion amongst some security researchers that the NIST standardised elliptic curves may have been backdoored by the NSA. Each curve contains an unexplained seed value from which coefficients are generated. The suspicion is that these seed values could have been created in such a way as to weaken the security of the curves they produce in an as-of-yet unknown way. This property is called elasticity.

As of Go 1.6, the standard library only has support for these NIST curves. There are some independently maintained implementations of other curves but the authors stress that they haven't been peer reviewed and shouldn't be used where security is paramount. Because of how the digital signature library is structured it will support additional curves without modification when they are introduced into the standard library.

Unit Tests

The digital signature library contains a comprehensive test suite. By using the MIFARE library it has been possible to write unit tests that do not depend on a physical card.

3.2.2 Difficulties Overcome

AID Swap Attack

During development of the digital signature library I discovered a vulnerability in my original proposal that would allow an adversary to manipulate the data whilst retaining a valid signature in certain circumstances.

To illustrate the attack, suppose that a card has two applications on it, A and B that are stored in sectors 1 and 2 respectively. The signature header specifies that the signature is across applications A and B and thus sectors 1 and 2, in that order.

Now suppose an adversary swaps the order of the applications in the signature header, and changes the application directory such that A now points to sector 2 and B points to sector 1. The signature is now across B and A (wrong way around), however this still resolves to sectors 1 and 2 (right way around) and thus the signature is still valid. If a reader attempts to read application A it will get the data for application B and vice versa.

To mitigate this, I altered the implementation such that the signature header is always included in the data that is signed, thus preventing an attacker from tampering with it.

Application Length Malleability Attack

Suppose that a card has two applications on it, A and B. A is stored in sectors 1 and 2 and B is stored in sector 3. If an attacker changes the application directory such that A points to sector 1 and B points to sectors 2 and 3 then the signature would still be valid.

To mitigate this attack I added the application length to the signature payload.

3.3 Revocation Simulation Suite

The simulation suite models a network of readers and simulates cards traveling through the network. It records how many cards travelled through each reader before a revoked UID was propagated to it.

3.3.1 Reader Network Topology

The network of readers is represented as a directed graph. Nodes represent readers and the directed edges from one node lead to all the readers that can be accessed after going through that reader. The indegree of a reader is the number of readers from which someone at this reader could have come from.

The simulation suite contains a generator that returns a semi-randomised topology based upon the number of offline and online readers. The indegree of each reader (the number of readers that it is accessible from) is assigned randomly from the range 1 to $\sqrt{num\_readers}$. The edges for each reader are then assigned randomly to satisfy the indegree. After this, a sanity check ensures that every reader has an edge to another one, if a reader doesn't then a new edge is added. This ensures that the simulation doesn't get stuck.

3.3.2 Card Simulation

The simulation is based on discrete time-steps. At each time-step every card moves to a new reader by choosing an available edge at random. The definition of a time-step is left intentionally vague, as any meaningful comparison with actual time is very much dependent on the frequency with which cards are used in the network. Some deployments will have cards that typically tap a dozen readers a day and others will have cards that stay dormant for months.

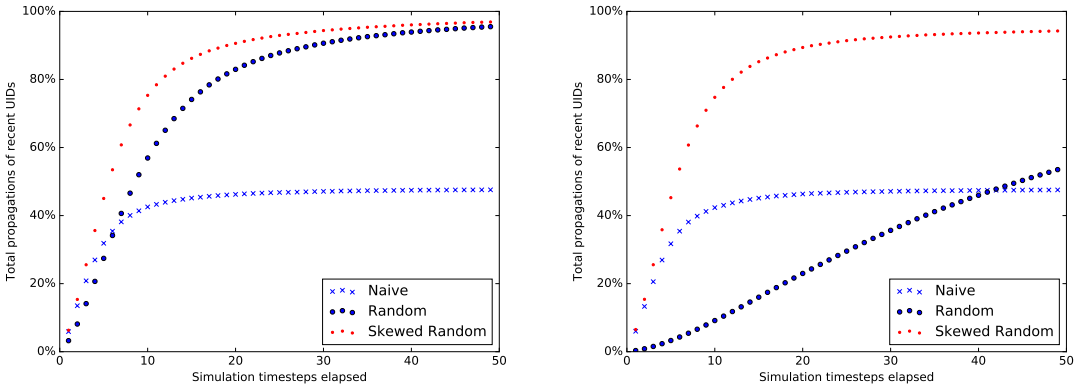
3.4 Revocation Gossip Protocol

The gossip protocol* enables the distribution of revoked card UIDs to offline readers by using the spare space on the cards. The development of the revocation gossip protocol was iterative. At each stage of development, the simulation suite identified specific areas of weakness, thus informing further iterations. Given the explorative nature of development, this section outlines each implementation in detail rather than the one which performed best. This helps to give context and justify the motivating factors behind each change made.

Naïve Implementation

The naïve implementation involves online readers storing a list of the revoked UIDs on the card. When the card is tapped on an offline reader the reader updates its internal list of revoked UIDs. When the number of revoked UIDs exceeds the number that can be stored on the card the most recently revoked ones are used.

*A gossip protocol is a type of computer-to-computer communication protocol that disseminates information throughout a network in a manner much like how rumours are spread throughout an office.



(a) Performance with 25 existing UIDs (b) Performance with 500 existing UIDs

Figure 3.6: Performance with varying numbers of fully propagated UIDs

This implementation is clearly not perfect, the revocation sector on the card has a restricted amount of storage. If more cards are revoked at one time than can be stored in the revocation sector, this implementation will fail to distribute some of them. The implementation is useful for assessing the performance of others. When the rate at which cards are revoked is sufficiently low the naïve approach is close to being optimal and thus provides a convenient baseline to indicate the inherent difficulty of a particular simulation configuration.

Random Implementation

Each time a card is tapped on an online reader this implementation chooses a random subset of the revoked card UIDs to place on the card.

The random implementation performs very well when the total number of revoked cards is low, however as can be seen from Figure 3.6, as the number of revoked cards increases, the performance decreases. This is because the probability that any given UID will be placed on a card is inversely proportional to the total number of revoked cards. As the total number of revoked cards increases, each new revoked UID is placed onto fewer cards with an increasing amount of the storage on each card being used for UIDs that have already been propagated throughout the network.

Skewed Random Implementation

The random implementation performed well initially but the simulation suite demonstrated that as the total number of revoked cards grew, its performance suffered. To improve its performance with a large number of revoked cards it is necessary to ensure that the probability of a recently revoked UID being placed on a card is independent of the total number of revoked cards.

The skewed implementation does this by prioritising UIDs that have not been placed

onto 5 cards. Each online reader maintains two buckets (arrays) of UIDs and a count of how many times each UID has been distributed. When a UID has been distributed 5 times, it is moved from the first bucket to the second. UIDs in the first bucket are always chosen before those in the second.

Chapter 4

Evaluation

The original aim of the project, as outlined in the proposal, was to develop a high-level MIFARE library, a MIFARE digital signature library, and a revocation gossip protocol for distributing revoked UIDs. The end result both encompasses the original goals, and additionally includes a simulation suite for simulating arbitrary networks of readers and cards. Whilst fulfilling the success criteria, the project has broadened slightly in scope, including additional countermeasures such as key diversification and emulated card detection, that further enhance security.

4.1 MIFARE Library

Development of the MIFARE library followed software-engineering best practices such as source code management (git) and comprehensive test coverage. The unit tests enabled me to catch several subtle mistakes which would have led to difficult to track down bugs, including one in a third party library. The library is structured to support future additions without breaking backwards compatibility. This would allow, for example, someone to easily add support for MIFARE Desfire cards.

The abstraction of the reader and card interfaces from their implementations was not only instrumental to the success of this project, but will also be of use to others writing software for interacting with MIFARE Classic cards. Any software written on top of the library can benefit from the mock driver to write automated unit tests rather than relying on human and hardware based verification. The library will be open sourced and thus constitutes a significant contribution to the community.

4.1.1 Choice of Language

Go proved to be an excellent choice of language to implement the library in. Making the library cross-platform has required no effort on my part and the standard library that Go provides has been very useful, in particular its cryptographic packages and test framework.

One of the original motivating factors behind the choice to use Go was its concurrency primitives and how they could be useful with the asynchronous nature of NFC communication. Whilst at a low level the communication with a card is asynchronous, I realised that it doesn't make sense to expose this in the library's API. The concurrency primitives were utilised in the simulation suite and enabled a significant speedup of the CPU-bound simulations by running them in parallel.

4.2 Digital Signature Protection

The digital signature library is built on top of the MIFARE library and uses the provided mock driver in its unit tests. The library provides methods for provisioning a signature, reading signed data, and writing signed data.

To both verify that the library worked and serve as a demo, I wrote a small command line application for interacting with some physical MIFARE 4K cards. The application simply stores and retrieves a username, it has three commands:

- `provision [username]`
  Allocates a sector for the username and a sector for the signature if not already allocated and writes the specified username.
- `sign`
  Signs the username and UID.
- `read`
  Reads the username and specifies whether the signature was valid.

The listing below shows the output when provisioning a new card. The signature is only valid after signing the username and UID.

```
$ ./demo/digsig provision alice
Provisioning card with UID: 50492583
Card provisioned

$ ./demo/digsig read
Name: alice [invalid]

$ ./demo/digsig sign

$ ./demo/digsig read
Name: alice [valid]
```

The commands below use the MIFARE offline cracker* to extract the secret keys from Alice's card and dump the data. The dump is then written to a different card. As this card has a different UID the signature is not valid.

```
$ mfoc -O alice.mfd
...
$ nfc-mfclassic w B alice.mfd demo.mfd
$ ./demo/digsig read
Name: alice [invalid]
$
```

*https://github.com/nfc-tools/mfoc

4.3 Ability to Mitigate Attack

It was never the aim of this project to completely "secure" MIFARE Classic cards. The security of the cards is fundamentally compromised. As is concluded in prior research[1] new deployments of NFC cards should use one of the several more secure alternatives such as the MIFARE DESFIRE EV2.

Organisations with existing deployments of MIFARE Classic cards should still plan to upgrade to a more secure card. The process of upgrading a large deployment such as the one at the UNIVERSITY OF CAMBRIDGE is a complex and costly one, both financially and in terms of time. All readers would have to be upgraded to support the new type of card before any cards can be issued. At the UNIVERSITY OF CAMBRIDGE this includes readers belonging to several distinct departments and colleges, each of which has a different system.

Until an NFC deployment can be upgraded or whilst it is ongoing, the countermeasures outlined in this project significantly enhance security. Without these countermeasures, an attacker need not have any technical expertise to mount an attack as software for cracking the cards is freely available. With the ease in which a malicious actor can access the tools required it would be surprising if the vulnerabilities weren't being actively exploited.

4.3.1 Impact on Attacks

Forge a card

With knowledge of the structure of the data on one card an attacker can produce a different card. For example, at the University of Cambridge an attacker can create a new card with an arbitrary CRSID.

Without the countermeasures there is no protection against this.

With the digital signature countermeasure an attacker would need to also forge the signature and therefore this attack is prevented.

Clone a card

With access to a card an attacker can make a copy of the card.

Without the countermeasures an attacker can extract the secret keys using freely available open source tools. This then allows them to clone any card to a blank card.

The key diversification countermeasure means an attack must extract the secret keys from every card they wish to clone which takes several minutes. With the digital signature countermeasure an attacker must either have a card that allows the UID to be written or have specialist equipment that can emulate the MIFARE

Classic card. The emulation detection countermeasure additionally requires that any specialist equipment implement the same PRNG as the MIFARE card to avoid detection. Even after this, a cloned card may be detected as the decrement counter will get out of sync.

Use a stolen card

Without the countermeasures the card will stop working with networked readers when it is reported stolen. The card will continue to work with offline readers unless they are manually updated.

With the countermeasures the card will stop working with networked readers when it is reported stolen. The revocation gossip protocol will also distribute the UID to the offline readers to prevent use with them after a short period of time.

The key diversification countermeasure prevents an attacker using the secret keys from one card to access data on another, thus an attacker is required to have prolonged access to a card to access the data.

4.4 Applicability to Other Cards

Whilst the focus of this project was on MIFARE Classic cards, it is far from irrelevant to other NFC cards. Many of the countermeasures outlined, such as digital signatures and the revocation gossip protocol are not inherently specific to MIFARE Classic cards. The use of digital signatures with other NFC cards would provide *defence in depth* — a well accepted security principle in which security measures are layered upon each other to provide redundancy when one security measure fails.

The revocation gossip protocol would be useful in any deployment with offline readers. The protocol could be altered to distribute other data such as reader configuration, allowing an organisation to update the configuration of offline readers.

4.5 Revocation Gossip Protocol

The goal of the revocation gossip protocol was to distribute the list of revoked card UIDs from online readers to offline readers. This allows an organisation to more effectively block the use of lost or stolen cards.

The simulation suite supports simulating a wide range of different network layouts. It was used throughout the development of the gossip protocol to identify performance issues and inform changes to the algorithm. In this section, the performance of the revocation gossip protocol is assessed against a range of different simulations.

4.5.1 Best-case performance

If just a single UID is revoked, then every card that is tapped on an online reader will contain the UID in its revocation sector. Each of the algorithms is optimal in this case, and thus it serves as a useful baseline for the inherent difficulty of a particular network of readers.

The simulation records the number of cards that tapped on each offline reader before the revoked UID was propagated to it. Figure 4.1 shows the best-case performance on a semi-randomised network (see Section 3.3.1) consisting of 500 offline readers and 100 online readers. The graph plots the 50th, 90th, and 99th percentile of card taps for varying numbers of cards per reader.

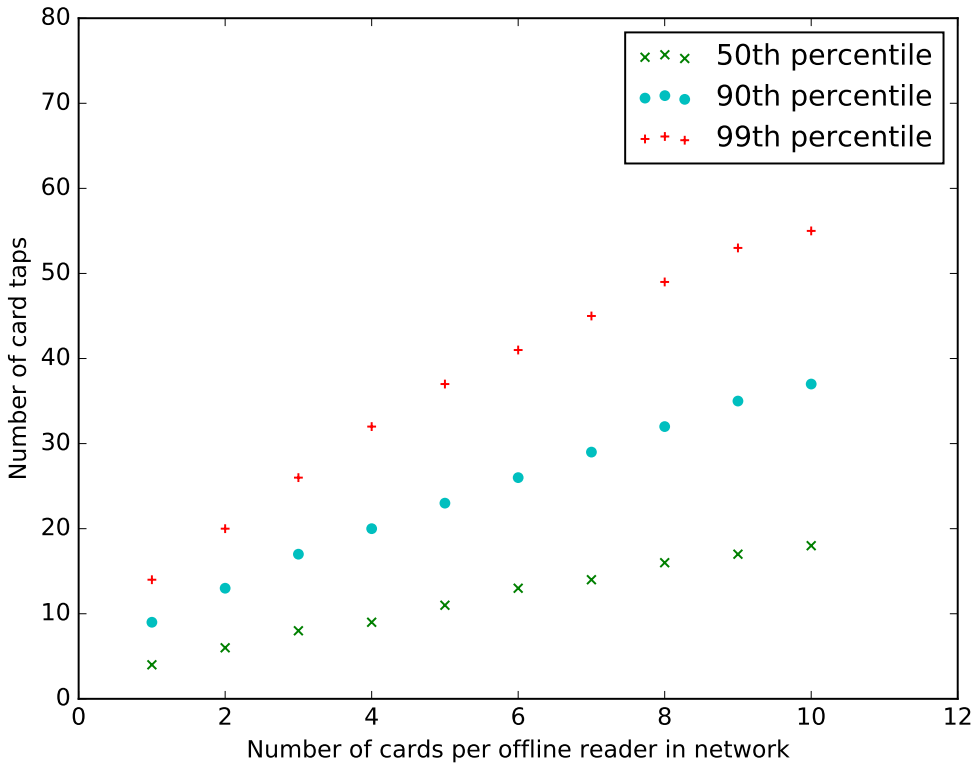


Figure 4.1: Card taps to update readers with revoked UID (500 readers)

It can be seen that as the number of cards increases, it takes more taps on average to propagate the UID to each reader. With 500 cards (one per offline reader), 99% of readers are updated in 14 taps or less. With 5000 cards (ten per offline reader), 99% of readers are updated in 55 taps or less.

From first glance it may seem that the best case performance is negatively impacted by increasing the number of cards. This is, however, not the case, as increasing the number of cards also increases the frequency of card taps on a reader. Figure 4.2 shows that increasing the number of cards decreases the number of simulation timesteps required to propagate the UID.

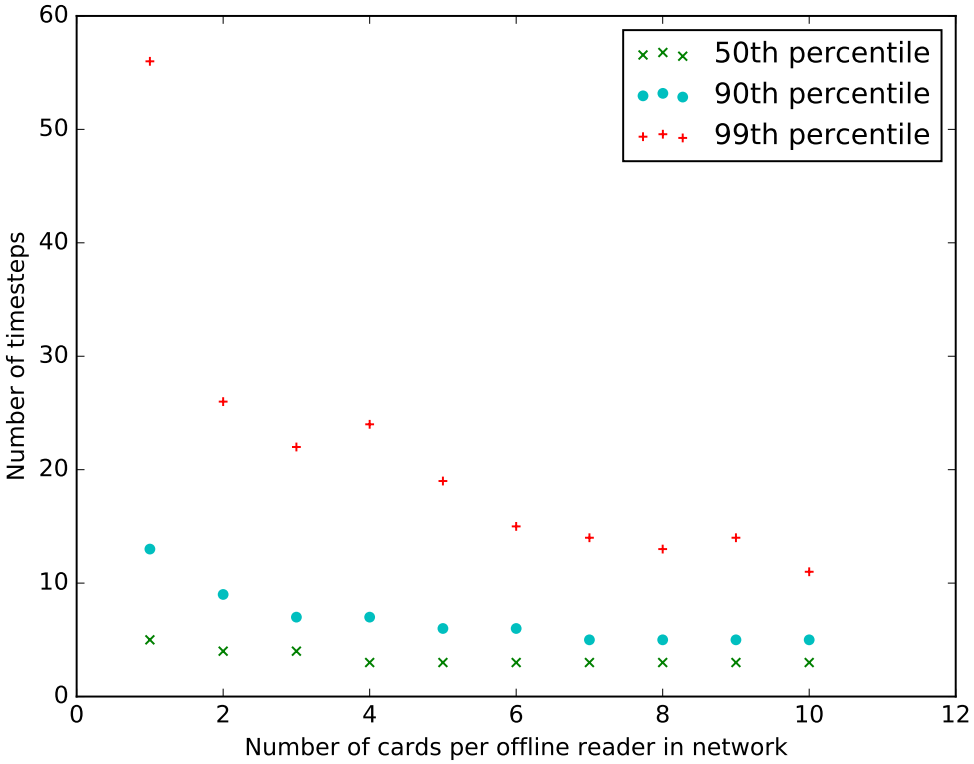


Figure 4.2: Timesteps to update readers with revoked UID (500 readers)

I'll reiterate that the absence of a link between a simulation timestep and a unit of actual time is intentional. The simulations can, within the assumptions of the model, determine how many taps and timsteps are required on each reader before the UID is propagated to it. They cannot determine how long this will take without making assumptions about how frequently cards are used. Any such assumption would be accurate for only a few networks, as the frequency with which cards are used can vary greatly between different card deployments. Within some deployments each card is used several times a day, where as in others a significant proportion of the cards may be dormant for months at a time.

Both Figure 4.1 and Figure 4.2 used a network with 500 readers. Simulations using 100 and 1000 readers produced similar results as shown in Figures 4.3 and 4.4. As expected, increasing the number of online readers improves performance as shown in Figure 4.5 on the following page.

4.5.2 Revoking several cards at once

If the number of revoked UIDs is less than the capacity of the revocation sector (12 UIDs) then each algorithm can simply place all revoked UIDs into the revocation sector. As previously mentioned, this gives a good indicator of the inherent difficulty of a network but does little to stress test each algorithm. If the number of revoked

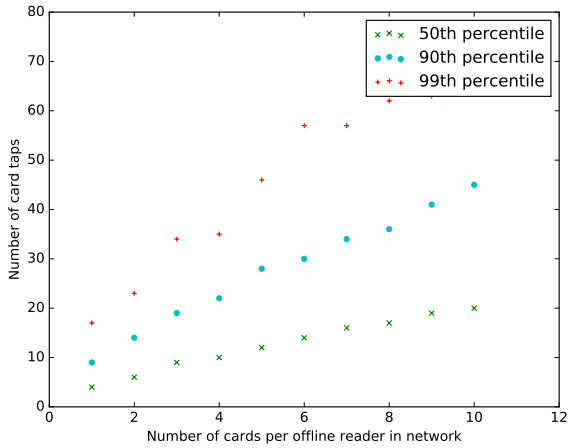


Figure 4.3: Card taps to propagate UID (100 offline and 10 online readers)

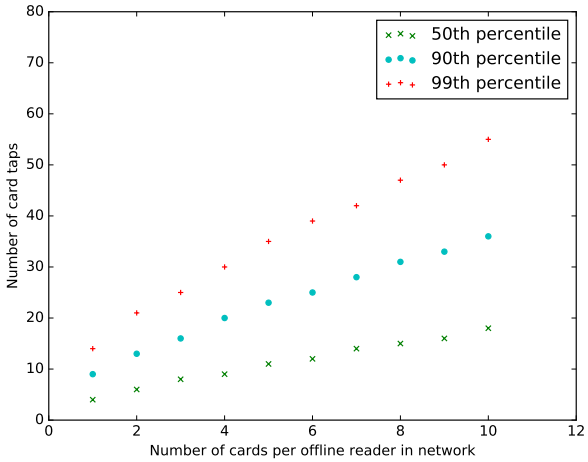


Figure 4.4: Card taps to propagate UID (1000 offline and 100 online readers)

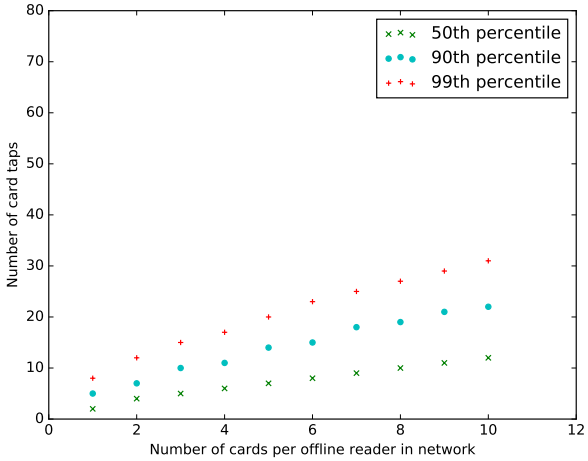


Figure 4.5: Card taps to propagate UID (500 offline and 500 online readers)

UIDs exceeds the capacity of the revocation sector, then the algorithm must choose a subset of the UIDs to place in the sector.

In this subsection I evaluate how well the naïve and random algorithms perform when several UIDs are revoked at once. The simulations use a network of 500 offline readers, 50 online readers, and 500 cards. For each timestep, the simulation records the percentage of propagations that have occurred. A propagation occurs when an offline reader sees a revoked UID for the first time and updates its internal list.

Figure 4.6 shows the relative performance of both the naïve algorithm and the random algorithm when thirteen UIDs (one more than the capacity of the revocation sector) are revoked at once. As is evident from the graph, the naïve algorithm completely fails to propagate one of the UIDs.

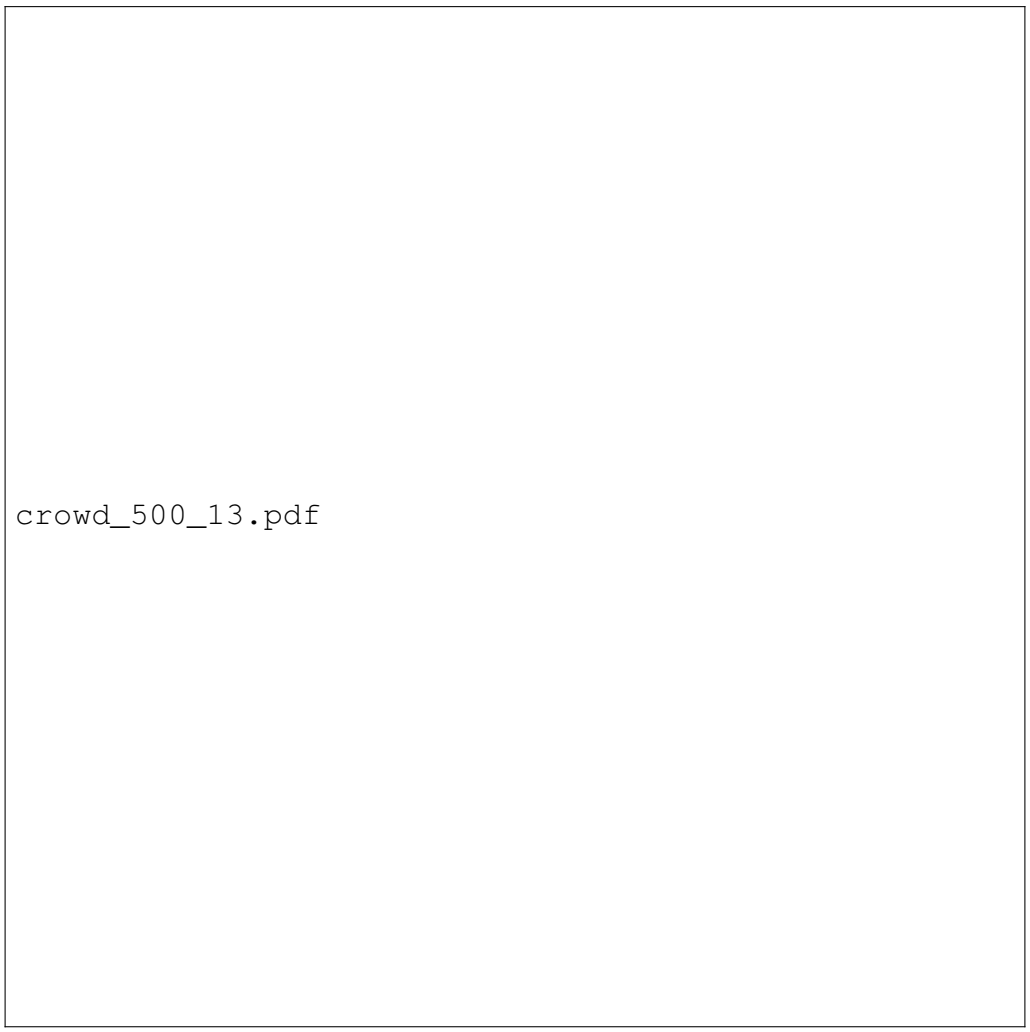
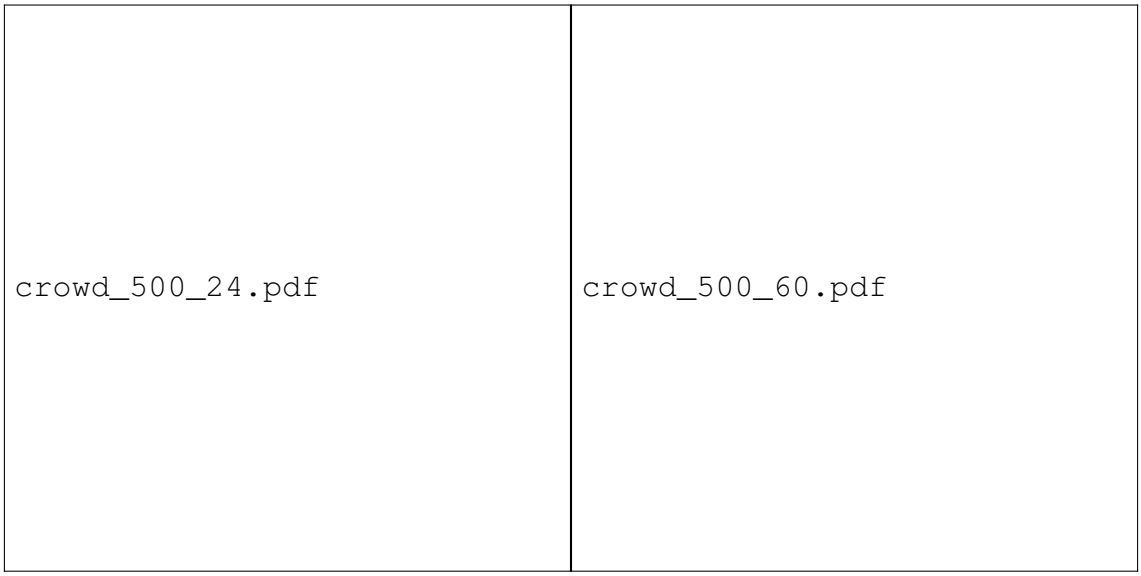


Figure 4.6: Propagation of 13 UIDs

The performance improvement of the random algorithm over the naïve algorithm increases as the number of revoked UIDs increases. Figure 4.7 on the following page shows the performance with 24 and 60 UIDs (2 and 5 times the revocation sector capacity respectively). From Figure 4.7b on the next page it can be seen that the

performance remains reasonable even when more than 10% of the total cards are revoked at once.



(a) Propagation of 24 UIDs (b) Propagation of 60 UIDs

Figure 4.7: Performance with varying numbers of UIDs

4.5.3 Increasing number of revoked cards

The total number of revoked UIDs increases over time. The random algorithm selects UIDs at random, irrespective of how long ago they were revoked. Over time, the proportion of all revoked UIDs that have been recently revoked decreases, and thus the recently revoked UIDs are placed on cards with decreasing frequency. The performance of the random algorithm, therefore, decreases as the total number of revoked cards increases.

As described in Section 3.4, the skewed random algorithm accounts for this by weighting the UIDs to ensure that recently revoked UIDs take precedent over those that have already been propagated. Figure 4.8 on the following page shows how well each algorithm propagates 25 UIDs with 100 existing revoked and fully propagated UIDs. Increasing the number of revoked UIDs has little effect on the skewed algorithm, as shown in Figure 4.9. As shown in Figure 4.9, increasing the number of revoked UIDs has a huge impact on the performance of the random algorithm but almost no impact on the skewed algorithm.

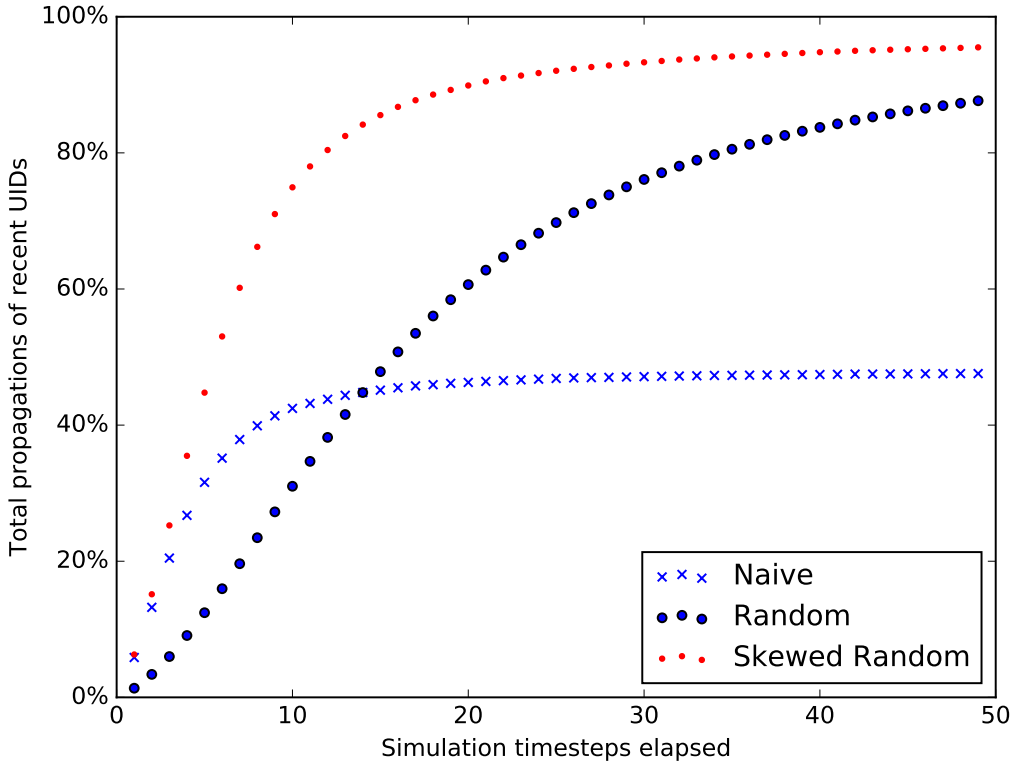
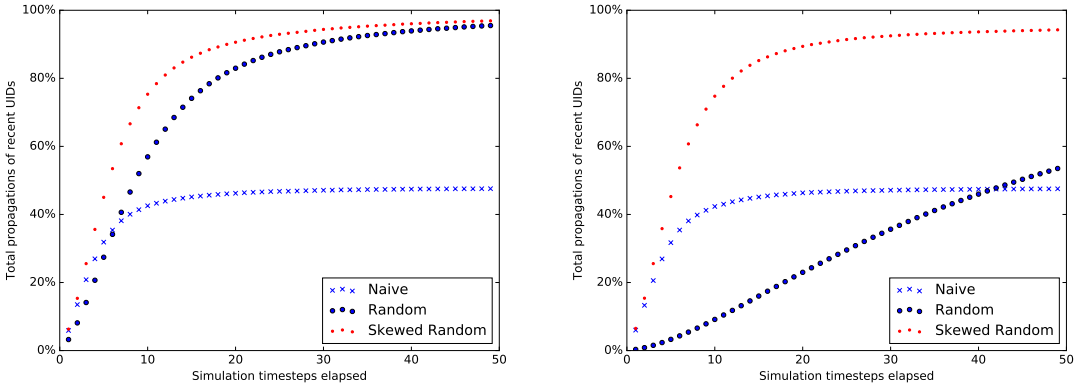


Figure 4.8: Propagation of 25 new UIDs with 100 fully propagated UIDs



(a) Performance with 25 existing UIDs (b) Performance with 500 existing UIDs

Figure 4.9: Performance with varying numbers of fully propagated UIDs

4.6 Simulation Suite

The simulation suite was instrumental to the success of the gossip protocol. Without the ability to simulate arbitrary networks, I wouldn't have had the data required to guide the changes (such as skewing). In this section, I evaluate the decisions I made and the performance of the simulation suite.

The simulation suite was wrapped in a command line application to make it easy to simulate different network configurations. The output of the CLI (as shown in Figure 4.10) contains all of the configuration values in the header. This allowed me to pipe the simulation output directly into a python script that generated appropriate graphs.

```
$ chain -offline=10 -alreadyRevoked=0 -newlyRevoked=1 -samples=2 -timesteps=5
# chain: 2, 5, 10, 0.100000, 1.000000, 0, 1
# sample: 0, rndSeed: 10
0: 0, 0, 0
1: 1, 1, 1
2: 2, 2, 2
3: 4, 4, 4
4: 7, 7, 7
# sample: 1, rndSeed: 11
0: 0, 0, 0
1: 1, 1, 1
2: 2, 2, 2
3: 3, 3, 3
4: 4, 4, 4
$
```

Figure 4.10: Output from simulation suite

4.6.1 Choice of language

When benchmarking the performance of each algorithm, the simulation is run on 10 different semi-randomised networks to ensure that the results are not specific to a single network. The extensive concurrency support that the Go programming language provides has been useful for optimising the performance of the simulation suite. Go's goroutines makes it easy to run these simulations in parallel, making full use of all the CPU cores available.

4.6.2 Performance

Several optimisations were made to improve the performance of the simulation suite. The mock driver in my MIFARE library contains a software implementation of all the MIFARE Classic card logic. After using the Go profiler it became apparent that the majority of the running time was spent carrying out these operations rather than doing the actual simulation. To reduce the running time, I implemented a fake driver that only supported reading and writing the revocation sector, and did not

do any access condition verification. The fake driver reduced the running time by a factor of five.

The running time is of order $\mathcal{O}(x \times y)$ where x is the number of simulation timesteps and y is the number of cards. Since each timestep involves moving each card to a new reader, the order cannot be reduced. Each timestep of the simulation has two parts; the first part moves each card and taps it on the new reader, the second part then queries each reader to collect the simulation metrics for that timestep.

Chapter 5

Conclusions

This project has explored the weaknesses of MIFARE Classic cards and detailed several countermeasures to mitigate the risk of attack. This project is of particular relevance to the University of Cambridge, which issues MIFARE Classic cards to all students and staff.

This project was a success. I have fulfilled the original aims of the project by developing a high-level MIFARE library for interacting with MIFARE cards, a MIFARE digital signature library for reading and writing signed data, and a gossip protocol for propagating revoked UIDs throughout the network. My research during the project gave me a thorough understanding of the weaknesses in the card, enabling me to devise several additional countermeasures. Combining these countermeasures significantly increases the resources required to mount an attack.

During the evaluation of the revocation gossip protocol, I demonstrated that it performs well on networks ranging from hundreds to thousands of readers. On a network the size of Cambridge's (albeit randomised) with one online reader for every two offline readers, 99% of readers are updated in 30 taps or less. Given the frequency with which readers are used, this would not take very long.

The focus of the project changed slightly since its inception, shifting more towards the gossip protocol and simulation suite, and away from the digital signature protection. With the benefit of hindsight I would have tried to collect some real-world card usage data. This would enable me to assess how well the simulation suite matches the real world.

An organisation with an existing MIFARE Classic deployment should upgrade to a more secure card, such as the MIFARE DESFIRE EV2. It may take a long time to update all readers, during the transition the countermeasures outlined in this project can be used to mitigate the risk of attack.

Kerckhoff's principle is as apt today as it was in the nineteenth century.

> A cryptographic system should be secure even if everything about the system, except the key, is public knowledge.

Bibliography

[1] F. D. Garcia, G. de Koning Gans, R. Muijrers, P. Van Rossum, R. Verdult, R. W. Schreur, and B. Jacobs, "Dismantling MIFARE classic," in *Computer Security-ESORICS 2008*, pp. 97–114, Springer, 2008.

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

[3] N. Courtois, K. Nohl, and S. O'Neil, "Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards.," *IACR Cryptology ePrint Archive*, vol. 2008, p. 166, 2008.

[4] K. Nohl and H. Plötz, "Mifare: Little Security, Despite Obscurity," in *24th Chaos Communication Congress*, December 2007.

[5] T. Rosati and G. Zaverucha, "Elliptic curve certificates and signatures for NFC signature records," *Research in Motion, Certicom Research*, p. 10, 2011.

[6] M. Kilas, "Digital signatures on NFC tags," *(Unpublished Master of Science). KTH Royal Institute of Technology, Sweden*, 2009.

[7] Philips Semiconductors, "Mifare Standard 4K Byte Card IC MF1 IC S70 Functional Specification," 2002.

[8] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA–a cost-optimized parallel code breaker," in *Proceedings of the 8th international conference on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, 2006.

UNIVERSITY OF CAMBRIDGE

Daniel Chatfield
Robinson College
dc584

Part II Project Proposal

Elliptic Curve Digital Signature Protection for Mifare Classic RFID Cards

October 22, 2015

supervised by

Dr. Markus Kuhn

Introduction

Many organizations use contactless smart cards for authentication, including the University of Cambridge. The Mifare Classic card is one of the most popular smart cards and is currently the card used by the University of Cambridge for everything from access to buildings to paying for meals.

The communication between the card and the reader is based on the ISO-14443-A standard with a proprietary authentication scheme that provides mutual authentication between the reader and the card. The proprietary authentication scheme has been the subject of academic research for several years. Initial research focussed on reverse engineering the protocol and then later research exploited weaknesses to fully compromise the cards [1].

The storage on the card is split into sectors, each sector has two keys associated with it (A and B) each of which can be configured to allow read/write access permissions. A typical deployment (like Cambridge University cards) uses one key as a read/write key that is kept secret by the card issuing office and the other key for reading that is distributed to the various organzations that need to read the cards.

Today it is possible to extract all the secret keys from a Mifare Classic card within 10 minutes if just one of the sectors uses a default key*. Since it is unusual for all the sectors to have unique keys it is usually much quicker. Once the keys have been extracted, the attacker can read/write to the card and any other cards that have the same keys.

The publication of these vulnerabilities prompted some organizations to move to more secure contactless smart cards but many are still using Mifare Classic cards due to the cost or difficulty involved in replacing all issued cards or upgrading readers.

I plan to improve the security of these cards by digitally signing the data along with the card UID. The reader would then verify that the signature is valid for the data on the card, thus preventing unauthorized modification. By including the card UID in the payload to be signed it substantially raises the bar for cloning a card as it would require the possession of either a Mifare Classic compatible card that allowed the UID to be set or a device that could emulate one such as a Proxmark 3. Elliptic curve digital signatures are appropriate as they offer much shorter signatures than traditional digital signature algorithms, this is desirable with the memory constraints of the card.

It is common to have the need to disable a card before it expires (for example, when an employee leaves or when a card is lost). Some door-access readers are "online"† and thus all that is required to revoke access to these is to remove the card from the access list, however a large proportion are "offline"‡ and require a manual update to the reader to block access to a specific card. I will design and implement a protocol

*The Mifare specification requires this for the first sector as it should contain data for identifying who issued the card.

†They communicate with some central server to establish whether they should grant access.

for using the cards themselves as a medium for propagating the list of revoked cards from online readers to offline readers.

Starting Point

I will draw upon the following during my project:

- **Computer Science Tripos**
  My project will use knowledge from parts 1A and 1B of the tripos. The digital signature implementation will draw from content from the Security I course. Theory from the 1B Networking course will be useful when designing the offline revocation list distribution. The part II Computer Systems Modeling course may prove useful for modeling the efficiency and reliability of my distribution protocol in different environments.
- **Experience**
  I have some experience with Mifare Classic cards and NFC more broadly and extensive experience with the Go programming language.
- **Go crypto libraries**
  Go has comprehensive cryptography libraries that implement much of the required elliptic curve cryptography required for this project.

Project Structure

Core library in Go

As well as supporting authenticating to, reading from and writing to Mifare Classic cards, the core library will support the writing and verification of elliptic curve digital signatures allowing a reader to perform offline verification of a card without holding the private key used to sign it.

I have chosen Go as the language to implement the core library as it compiles to statically linked binaries that don't require a runtime, has excellent support for cryptographic operations and its channels are a natural fit for the asynchronous nature of NFC communication.

Go can be easily cross compiled for different platforms including Android and iOS, making the codebase very portable.

The development of the library will follow software engineering best practices. I will develop a comprehensive test suite comprising of unit tests to test individual methods, and acceptance tests that test high-level functions like writing an entire

[‡]They either have a preprogrammed access list or the card has a signed token.

card and then reading it. This will help ensure correctness and reduce the likelihood of regressions being introduced as features are added.

The library will provide abstractions to make it easy to perform common tasks in third party projects, without being familiar with how NFC works.

Command Line Interface

The library will be wrapped in a command line interface that exposes its high-level functionality. The CLI will have two modes — one suitable for consumption by a human using a terminal, and one that returns machine readable JSON to make it easy for projects not written in Go to use the library without attempting to both parse the text output and rely on it being backwards compatible in future versions.

Desktop Application

A desktop application will sit on top of the CLI providing a graphical interface for managing the lifecycle of the cards including the writing, reading and verification. This desktop application will be developed using Github's Electron [2], as this makes it easy to make cross-platform applications with rich UIs.

Offline Revocation List Propagation Protocol

The development of the offline propagation protocol will be iterative. I will first make a simple and naïve protocol and create a simulation suite that benchmarks this protocol in different environments*. I will then evolve the design to overcome weaknesses, using the simulation suite results to justify and drive the changes made.

Possible Extensions

I'm sure that further possible extensions will become apparent whilst carrying out the project but two possibilities are:

Stronger protection of the private key

The security of the system relies on the private key remaining private. To help ensure this I could extend the application to support the use of a hardware security

*I will vary the number of cards and readers (including offline/online split). I will also simulate different card access patterns in an attempt to both simulate likely real world usage and also identify pathological scenarios.

module that would prevent the key from being covertly extracted from one of the machines that sign the cards.

Alternatively, I could allow the application to connect to a central server to do the signing. This would allow an organization to distribute the application to trusted employees without entrusting them with the key as they would have to sign in to the central server* which could then log what was signed by which employee and access could be revoked immediately.

Offline access violation reporting

As I have already stated, many readers are offline and cannot report access violations. Typically the only way to discover such violations is for a technician to physically download the access log† from the reader. I could extend my project to allow these doors to use the cards as a medium to report these violations back to an online reader.

Success Criteria

At the end of the project I should be able to:

- Generate new public/private keypairs for use with a new card deployment (or upgrading an existing deployment to support signatures).
- Load an existing private key or public key.
- Provision a new card with a digital signature.
- Upgrade an existing card to include a digital signature.
- Read a card with a digital signature and verify that the signature matches.
- Reject cloned cards where the UID hasn't been spoofed.
- Revoke a card and use the cards to propagate this to offline readers.

Project Timetable

Michaelmas term

24th Oct — 8th Nov

1. Obtain an NFC reader.
2. Obtain some blank Mifare Classic cards.

*Cambridge University could use raven for this purpose
†Not all readers store a log

3. Research elliptic curve cryptography.
4. Familiarize myself with libnfc.

9th Nov — 22nd Nov

1. Research published weaknesses in Mifare Classic cards.
2. Develop a threat model against the existing system to use as a benchmark for comparison.
3. Begin development of the core library.

Milestone 1 By 22nd November I should be able to read and write to a Mifare card using my library.

23rd Nov — 6th Dec

1. Develop the command line wrapper around the library.
2. Test the signature creation and verification using Mifare cards.

Christmas vacation

6th Dec — 10th Jan
Start development of the desktop application.

Lent term

11th Jan — 17th Jan
Write progress report and complete desktop application.

Milestone 2 By 17th January I should be able to use the desktop application to read and write to the cards and verify the digital signatures.

18th Jan — 31st Jan

1. Design a protocol for offline propagation of the card revocation list.
2. Use modeling to verify that the design works at scale.

1st Feb — 14th Feb
Implement offline revocation list propagation.

15th Feb — 13th Mar
Main chapters of dissertation

Easter Vacation

14th Mar — 17th Apr
Proof read dissertation.

Milestone 3 By the end of the Easter vacation I should have delivered the full draft proposal to my supervisor and DoS.

Resources Required

To complete the project I will need an ISO-14443-A compatible USB NFC reader and some blank Mifare Classic cards. I don't envisage any problems sourcing these as they are both widely available at relatively low cost.

Development and testing will be done on my 2013 MacBook Pro, I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Backup Policy

I will use git for all source code (including latex), using either github or bitbucket as a remote. In addition to this, I will use Apple time machine to periodically backup all files on my laptop. In the event of hardware failure, I will only lose any changes made since the most recent git push or time machine backup which should be no more than a days work.

Any computer that supports compiling Go is suitable for development and therefore should my laptop become permanently broken it will be easy to source a replacement.

References

[1] W. H. Tan, "Practical attacks on the mifare classic," *Imperial College London*, 2009.

[2] GitHub, "Electron," 2015. [Online; accessed 15-October-2015].