Logical Formalisation and Analysis of the Mifare Classic Card in PVS

Bart Jacobs and Ronny Wichers Schreur

Institute for Computing and Information Sciences, Radboud University Nijmegen
Heijendaalseweg 135, 6525AJ Nijmegen, The Netherlands
{bart,ronny}@cs.ru.nl

Abstract The way that Mifare Classic smart cards work has been uncovered recently [2,4] and several vulnerabilities and exploits have emerged. This paper gives a precise logical formalisation of the essentials of the Mifare Classic card, in the language of a theorem prover (PVS). The formalisation covers the LFSR, the filter function and (parts of) the authentication protocol, thus serving as precise documentation of the card's ingredients and their properties. Additionally, the mathematics is described that makes two key-retrieval attacks from [2] work.

1 Introduction

Theorem provers provide machine support for the formalisation and verification of systems and their properties. They are used both for hardware and for software. A theorem prover may be seen as a sceptical colleague that checks and documents all individual proof steps and helps with tedious details. There are several sophisticated interactive theorem provers around, such as Isabelle [3,7], Coq [6], NQTHM [1] and PVS [5,9].

In this paper we (happen to) use PVS. But we do not rely on any special property or power of PVS. We shall try to abstract away from the specifics of PVS, and formulate results in the language of (dependently) typed higher-order logic, using a certain level of pretty printing. The point we wish to make is that using a theorem prover is useful (also) in the area of computer security, for a precise description and analysis of one's system. As such it may be used as part of precise documentation, or even as part of a certification procedure. As will be illustrated, this works well for relatively unsophisticated systems, like smart cards with low-level operations. The PVS formalisation is available on the web [1].

The formalisation presented in this paper is specific to the Mifare Classic card and so it does not carry over to other systems. There is little or no uniformity in (proprietory) cryptographic systems, and hence there can be no uniformity in their formalisations. In a broader context this paper sets out to show that formalisations contribute to the documentation and analysis of low-level security protocols. The method as such does apply to other systems.

[1] http://www.cs.ru.nl/~ronny/mifare-pvs

The Mifare Classic is a contactless (RFID) smart card, sold by NXP (formerly Philips Semiconductors), that is heavily used in access control and public transport (like in London's Oyster card, Boston's Charlie card, and the Dutch OV-chipkaart). It is estimated that over 1 billion copies have been sold worldwide. The design goes back to the early nineties, predating Common Criteria evaluation practices in the smart-card area. The card (and Mifare readers) contains a proprietary encryption algorithm, called Crypto1. It uses a 48-bit linear feedback shift register (LFSR), a device that is well-studied in the literature (see *e.g.*, [8,10]), together with a special filter function that produces the keystream bits.

The security of the card relies partly on the secrecy of this algorithm. Details of Crypto1 have emerged, first after hardware analysis [4][2], and a bit later[3] after a cryptanalysis [2]. The latter reference presents a mathematical model of the card, together with several attack scenarios for key retrieval. The current paper builds on [2] and elaborates certain mathematical (logical) details of this model and these attacks. It does not add new (cryptographic) results, but provides further clarity about the card. In doing so it points out where design flaws reside and how they can be exploited.

The paper is organised as follows. It first describes some general properties of LFSRs and filter functions, focusing on what is relevant here, and not developing much meta-theory. Subsequently, Section 3 gives the crucial ingredients that model the stream cipher Crypto1 of the Mifare Classic card, and Section 4 shows how these operations can be rolled back. Section 5 illustrates how the definitions and results from Section 3 establish (part of) the correctness of the mutual authentication protocol between a card and a reader. Finally, Section 6 elaborates the mathematical properties underlying two attacks from [2], namely the "two-table" and "odd-from-even" attacks.

2 Shift registers, generally

This section describes the formalisation of shift registers in PVS, together with some basic properties. The feedback function will at this stage be a parameter. A concrete version will be provided in Section 3.

The formalisation uses the PVS type `bvec[N]` of bit vectors of length `N:nat`. The natural number `N` is thus a parameter. The type may be instantiated concretely as `bvec[10]`, which yields the type of bit vectors of length 10. The type `bvec[N]` is defined as the type of functions from natural numbers below `N` to the type `bit = bool`, with **TRUE** and **FALSE** as (only) inhabitants. One writes `fill[N](b):bvec[N]` for the constant bit vector filled with `b:bit` at every position.

The logical description of LFSRs at this stage contains two parameters, namely their length `LfsrSize:posnat` (a positive natural number) and a feedback function `feedback:[bvec[LfsrSize]→bit]` that maps a bit vector of length

[2] Made public in a presentation at the Chaos Computer Club, Berlin, 27/12/07.
[3] In a letter of the Dutch Interior Minister to Parliament of 12/3/08.

LfsrSize to a bit. For convenience we abbreviate this type of bit vectors as 'state' in a PVS type definition:

```
state : TYPE = bvec[LfsrSize]
```

Then we can define the basic "left shift" function that captures the operation of an LFSR. It is called shift1in because it takes one bit and puts it into the LFSR on the right, while shifting the whole LFSR one position to the left.

```
shift1in : [state, bit → state] =
   λ(r:state, b:bit) : λ(i:below(LfsrSize)) :
     IF i < LfsrSize - 1
     THEN r(i+1)                 % shift left
     ELSE b XOR feedback(r)      % put new value at i = LfsrSize - 1
     ENDIF
```

A picture of a concrete LFSR appears in Figure 1 in Section 3. Notice that the leftmost bit at position 0 is dropped, and that a new bit is inserted at the rightmost position LfsrSize-1. Via recursion an "N-ary" version is defined in a straightforward manner:

```
shiftNin : [state, N:nat, bv:bvec[N] → state] = ...
```

One can then prove basic properties, like:

```
shiftNin(r, N, bv)(i) = r(i+N)
shiftNin(shiftNin(r, N1, bv1), N2, bv2) = shiftNin(r, N1+N2, bv1 o bv2)
```

where i < LfsrSize-N and o is concatenation of bit vectors.

During initialisation a Mifare card and reader each feed certain (nonce) data into their LFSRs, see Section 5; afterwards they use their LFSR to produce a keystream by feeding it with 0s. This is captured by a special 'advance' function in PVS that has the number n:nat of inserted zeros as argument:

```
advance : [state, nat → state] =
  λ(r:state, n:nat) : shiftNin(r, n, fill[n](FALSE))
```

It forms an action with respect to the monoid of natural numbers since it satisfies:

```
advance(r, 0) = r      advance(r, n+m) = advance(advance(r,n), m)
```

2.1 Adding a filter function parameter

We remain a bit longer within the generic setting of LFSRs. We now add another parameter, namely a function filfun:[state→bit] that produces an output bit for an arbitrary state. A basic (single) step in the Mifare initialisation phase of card and reader involves processing one input bit while producing one (encrypted) output bit that is sent to the other side. There it is processed in a dual way, as described by the following two functions.

```
shiftinsend1 : [[state, bit] → [state,bit]] =
  λ(r:state, b:bit) : (shift1in(r,b), b XOR filfun(r))
receiveshiftin1 : [[state, bit] → state] =
  λ(r:state, b:bit) : shift1in(r, b XOR filfun(r))
```

These two functions satisfy the following "correctness" result.

```
∀(r:state, b:bit) :
  LET (r1,b1) = shiftinsend1(r,b) % b1 is transmitted, encrypted
  IN  receiveshiftin1(r,b1) = r1
```

This basic result requires some explanation: assume the two sides (card and reader) are in the same state `r` before performing these operations. Assume:

- one side (actually the reader) performs `shiftinsend1` and shifts one bit `b` into its state (leading to successor state `r1`), while transferring the encrypted version `b1 = b XOR filfun(r)` of `b` to the other side;
- the other side (the card) performs `receiveshiftin1` and receives this encrypted bit `b1`, decrypts it via `b1 XOR filfun(r)` and shifts it into its own state (which we assume to be equal `r`).

Then: both sides are again in the same post state, namely `r1`. Hence by performing these operations card and reader transfer data and remain in sync. In this way they communicate like via one-time pads, except that the keystream has cycles.

Also N-ary versions of the functions `shiftinsend1` and `receiveshiftin1` are defined, with appropriate properties. They are used in the following two functions

```
load_and_send_reader_nonce : [[state, nonce] → [state, nonce]] =
   λ(r:state, plain:nonce) : shiftinsendN(r, NonceSize, plain)
receive_reader_nonce : [state, nonce → state] =
  λ(r:state, cipher:nonce) : receiveshiftinN(r, NonceSize, cipher)
```

which will be used in the explanation of the Mifare authentication protocol in Section 5.

Of course, many more definitions and properties may be introduced for such abstract LFSRs. We confine ourselves to what is needed in our logical theory of the Mifare Classic. It includes a function to generate keystream bits, in the following way.

```
stream : [state, n:nat → bit] =
  λ(r:state, n:nat) :  filfun(advance(r,n))
```

It is used in a similar function `cipher` that not only produces keystream, but also the resulting state. It is defined with a dependent product type in:

```
cipher : [state, n:nat → [state, bvec[n]]] =
  λ(r:state, n:nat) : ( advance(r,n), λ(i:below(n)) : stream(r,i) )
```

It is well behaved, in the sense that it satisfies:

```
cipher(r, n+m) =
  LET (r1,c1) = cipher(r,n), (r2,c2) = cipher(r1,m) IN (r2, c2 o c1)
```

3 The Mifare LFSR and filter function

The parameters (like `LfsrSize`, `feedback` and `filfun`) that were used in the previous section are now turned into the specific values that they have in the Mifare Classic.

The sizes are easy:

```
LfsrSize : nat = 48     NonceSize : nat = 32
```

The Mifare feedback function, described as a generating polynomial, like in [4,2], is

$$g(x) = x^{48} + x^{43} + x^{39} + x^{38} + x^{36} + x^{34} + x^{33} + x^{31} + x^{29} + x^{24} + x^{23} + x^{21} + x^{19} + x^{13} + x^{9} + x^{7} + x^{6} + x^{5} + 1.$$

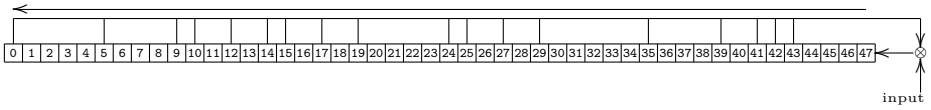

Figure 1. Mifare Classic LFSR.

In PVS this becomes, due to a reverse listing of entries:

```
MfCfeedback : [bvec[LfsrSize] → bit] =
  λ(r:bvec[LfsrSize]) :
    r(0) XOR r(5) XOR r(9) XOR r(10) XOR r(12) XOR r(14) XOR
    r(15) XOR r(17) XOR r(19) XOR r(24) XOR r(25) XOR r(27) XOR
    r(29) XOR r(35) XOR r(39) XOR r(41) XOR r(42) XOR r(43)
```

Notice that x^i corresponds to `r[48-i]`. The representation that we have chosen is the one that is most convenient in formulating definitions and properties. Now we can properly instantiate the theory of the previous section and obtain the type for "Mifare Classic LFSR" as:

```
MfClfsr : TYPE = state[LfsrSize, MfCfeedback]
```

We turn to the filter function for the Mifare Classic. It is constructed in several steps, via two auxiliary functions `MfCfilfunA` and `MfCfilfunB` that each produce one bit out of a 4-bit input. Such functions are usually described by 4 hexadecimal digits, capturing the conjunctive normal form. In this case we have `MfCfilfunA` = `0x26C7` and `MfCfilfunB` = `0x0DD3`, which can be simplified to a disjunctive normal form:

```
MfCfilfunA(b3, b2, b1, b0:bit) : bit =
  (  (¬b3 ∧ ¬b2 ∧ ¬b1) ∨ (b3 ∧ ¬b1 ∧ b0) ∨ (¬b2 ∧ b1 ∧ ¬b0) ∨ (¬b3 ∧ b2 ∧ b1)  )
MfCfilfunB(b3, b2, b1, b0:bit) : bit =
  (  (b3 ∧ ¬b2 ∧ ¬b0) ∨ (¬b3 ∧ b2 ∧ ¬b0) ∨ (b3 ∧ ¬b2 ∧ b1)
     ∨ (¬b3 ∧ b2 ∧ b1) ∨ (¬b3 ∧ ¬b2 ∧ ¬b1)  )
```

The LFSR and filter function of the Mifare Classic can now be depicted in Figure 2.

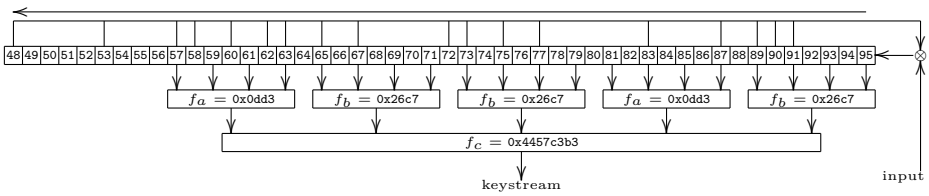

Figure 2. Crypto1.

One can then prove in PVS that these descriptions correspond to the conjunctive normal form given by the hexadecimal descriptions, but also to a "shift" description with the above values 0x26C7 and 0x0DD3: for a bit vector b of length 4,

```
MfCfilfunA(b(3), b(2), b(1), b(0))
  = right_shift(bv2nat(b), h2 o h6 o hC o h7)(0)
MfCfilfunB(b(3), b(2), b(1), b(0))
  = right_shift(bv2nat(b), h0 o hD o hD o h3)(0)
```

where bv2nat(b) gives the numerical value of the bit vector b, right_shift performs a number of shifts, as described by its first argument, and h2 etc. is the hexadecimal number 2, as bit vector of length 4 (with o describing concatenation, as before).

Two of these "A" and three "B" functions are combined into a new function that takes 20 bits input:

```
MfCfilfun20(b:bvec[20]) : bit =
  MfCfilfunC( MfCfilfunA(b(0), b(1), b(2), b(3)),
              MfCfilfunB(b(4), b(5), b(6), b(7)),
              MfCfilfunB(b(8), b(9), b(10), b(11)),
              MfCfilfunA(b(12), b(13), b(14), b(15)),
              MfCfilfunB(b(16), b(17), b(18), b(19)) )
```

where MfCfilfunC = 0x4457C3B3. Finally, the Mifare Classic filter function is described, following [2], as:

```
MfCfilfun(r:MfClfsr) : bit = MfCfilfun20(λ(i:below(20)) : r(9+2*i))
```

Important to note is the regularity of its application: on all odd positions 9, 11, 13, ..., 47. This regularity is one of the weaknesses of the Mifare Classic, which can be exploited in various ways as we shall see in Section 6.

Finally, here are some test results that are proven in PVS simply by a single proof command "grind".

```
MfCfilfun(hA o hE o hA o h6 o h1 o hC o h9 o hC o h1 o hB o hF o h0) = 1
MfCfilfun(h7 o hD o hA o h8 o h8 o h0 o h1 o h8 o h8 o h6 o h1 o h5) = 0
```

4 Rollback results

In this section we explore the structure described in the previous section, where we focus on the possibility of rolling back left shifts of the Mifare Classic register.

A first step is that we can recover the leftmost bit that is dropped in a single left shift step, if we know what the input bit is, via the following function.

```
leftmost : [MfClfsr, bit → bit] = λ(r:MfClfsr, b:bit) :
  r(47) XOR b XOR  % plus previous XORs, shifted one position
  r(4) XOR r(8) XOR r(9) XOR r(11) XOR r(13) XOR r(14) XOR
  r(16) XOR r(18) XOR r(23) XOR r(24) XOR r(26) XOR r(28) XOR
  r(34) XOR r(38) XOR r(40) XOR r(41) XOR r(42)
```

so that we can define an inverse of the `shift1in` function from Section 2 as:

```
shift1out : [MfClfsr, bit → MfClfsr] =
   λ(r:MfClfsr, b:bit) : λ(i:below(LfsrSize)) :
     IF i > 0
     THEN r(i-1)
     ELSE leftmost(r,b)      % at position i = 0
     ENDIF
```

and prove that they are indeed each other's inverses:

```
shift1out(shift1in(r, b), b) = r     shift1in(shift1out(r, b), b) = r
```

This shifting-out extends to an N-ary version, which is then inverse to N-ary shifting-in. Interestingly, the earlier advance function can now be extended from natural numbers to integers as:

```
Advance : [MfClfsr, int → MfClfsr] = λ(r:MfClfsr, n:int) :
    IF n ≥ 0
    THEN advance(r, n)
    ELSE shiftNout(r, -n, fill[-n](FALSE))
    ENDIF
```

We now get an action with respect to the monoid of integers:

```
Advance(r, 0) = r     Advance(r, i+j) = Advance(Advance(r,i), j)
```

where `i,j:int`. This allows us to smoothly compute a keystream not only in forward but also in backward direction.

4.1 Rolling back communication

So far we have concentrated on rolling back the LFSR. Since the Mifare Classic filter function `MfCfilfun` does not use the bit at position 0 it can also be reconstructed after a shift-left. This is another design error. We proceed as follows.

```
shiftout_MfCfilfun(r:MfClfsr) : bit =
  MfCfilfun20(λ(i:below(20)) : r(8+2*i))
shiftoutsend1 : [[MfClfsr, bit] → [MfClfsr,bit]] =
  λ(r1:MfClfsr, b1:bit) : LET b = b1 XOR shiftout_MfCfilfun(r1)
                          IN  (shift1out(r1,b), b)
```

Then we obtain an inverse to the basic step of the card from Subsection 2.1:

```
shiftoutsend1(shiftinsend1(r, b)) = (r, b)
shiftinsend1(shiftoutsend1(r, b)) = (r, b)
```

This can be done multiple times.

5 The Mifare Classic authentication protocol

When a card reader wants to access the information on a Mifare Card it must prove that it is allowed to do so. Conversely the card must prove that is a authentic card. Both security goals are accomplished by a mutual-authentication mechanism based on a symmetric-key cipher. Figure 3 pictures in detail how an authentication proceeds.

	Card		Reader
0		anti-c(uid) $\longrightarrow$	
1		auth(block) $\longleftarrow$	
2	$S_1 \leftarrow K_{\text{block}}$		$S_1 \leftarrow K_{\text{block}}$
3	picks n_C		
4	$S_2 \leftarrow \text{shiftinN}(S_1, \text{uid} \oplus n_C)$		
5		n_C $\longrightarrow$	
6			$S_2 \leftarrow \text{shiftinN}(S_1\ \text{uid} \oplus n_C)$
7			picks n_R
8			$(S_3, \{n_R\}) \leftarrow \text{send\_reader\_nonce}(S_2, n_R)$
9		$\{n_R\}$ $\longleftarrow$	
10	$S_3 \leftarrow \text{receive\_reader\_nonce}(S_2, \{n_R\})$		
11			$(S_4, \text{ks}_2) \leftarrow \text{cipher}(S_3)$
12		$\text{suc}^2(n_C) \oplus \text{ks}_2$ $\longleftarrow$	
13	$(S_4, \text{ks}_2) \leftarrow \text{cipher}(S_3)$		
14	verify $\text{suc}^2(n_C)$		
15	$(S_5, \text{ks}_3) \leftarrow \text{cipher}(S_4)$		
16		$\text{suc}^3(n_C) \oplus \text{ks}_3$ $\longrightarrow$	
17			$(S_5, \text{ks}_3) \leftarrow \text{cipher}(S_4)$
18			verify $\text{suc}^3(n_C)$

Figure 3. Mifare Classic Authentication Protocol.

Because Mifare Classic cards operate through radio waves, it is possible that more than one card is within range of a reader. To distinguish different cards, each card has a unique id that is send to the reader (step 0). This 32-bit uid also plays a role in the cipher.

The information on the Mifare Classic is divided into blocks. The reader starts an authentication session for a specific memory block (step 1). Each block is protected by a different 48-bit key that is known by both the card and the reader. Card and reader initialise their shift registers with this key K_{block} (step 2).

The card subsequently chooses a 32-bit challenge or card nonce (n_C). This card nonce is added ($\oplus$) to the uid and the result is fed into the LFSR. Also the card nonce is send to the reader in the clear, which then also feeds $n_C \oplus \mathsf{uid}$ into its LFSR.

Then it is up to the reader to pick a 32-bit reader nonce n_R. This nonce is also fed into the LFSR. After each bit the output of the filter function is collected in the encrypted reader nonce $\{n_R\}$ (send_reader_nonce in step 8).

Upon reception of the encrypted reader nonce $\{n_R\}$ the card performs the inverse operation of send_reader_nonce, that is receive_reader_nonce.

At this moment (after step 10) the cipher is initialised. The keystream now consists of the output of the filter function after each shift of the LFSR. All further communication is encrypted by adding the keystream to the clear text. Decryption is simply adding the keystream to the cipher text.

The reader responds to the card's challenge by sending the encrypted card nonce n_C, or rather the encryption of the expression $\mathsf{suc}^2(n_C)$. The function suc is actually computed by another 16-bit LFSR that is used to generate the card nonces. The card can decrypt the reader's response and verify that it corresponds to the expected result. This establishes that the reader can correctly encrypt the challenge, which presumably means that the reader has knowledge of K_{block} and thus is allowed to access that block.

To complete the mutual authentication, the card returns the encryption of $\mathsf{suc}^3(n_C)$. The reader can verify that the card's response is properly encrypted, which implicitly established the authenticity of the card.

To show that a reader and a card can perform a successful mutual authentication, we can show that after each step they are both in the same state. In Figure 3 this means that every S_n on the card side is equal to the S_n on the reader side. For most steps this is easy, since the card and the reader perform identical operations. Only when the reader nonce is processed, do the card and reader operate differently. The following PVS theorem states the correctness property of this step.

```
∀(s2 : state, plain_reader_nonce : nonce) :
  LET
      (rs3, encrypted_reader_nonce)
         = load_and_send_reader_nonce(s2, plain_reader_nonce),
      cs3
         = receive_reader_nonce(s2, encrypted_reader_nonce)
  IN
      cs3 = rs3
```

6 Formalising attacks

This section formalises the essentials of two attacks from [2]. They form a *post hoc* justification of the (C-code) implementation that underlies [2]. One can justifiably ask: what is the point of such a formalisation? After all, the attack in C can be executed and thus shows if it works or not. It does not need to work all the time, under all circumstances and only needs to work as a prototype[4], to show the feasibility of exploiting certain card vulnerabilities.

Our answer is that the formalisation explains the details—including assumptions and side-conditions—of the attacks and thus clearly demonstrates the precise vulnerabilities on which the attacks are built. This clarity may help to prevent or counter such vulnerabilities in similar situations.

6.1 The two-table attack

The first attack that will be formalised comes from [2, §§6.3]. It exploits the fact that the filter function `MfCfilfun` acts on only twenty LFSR positions, which are all at regular, odd positions (9, 11, ..., 47). Hence after shifting the LFSR two positions the filter functions gets very similar input.

This attack proceeds as follows. Assume we have a certain amount of keystream (at least 12 bits long). The aim is to find "solutions", namely LFSR states that produces this keystream, via the filter function. The first step is to define appropriate types for this setting:

```
keystream : TYPE =
  [# len : {n:posnat | even?(n) ∧ n ≥ 12}, bits : bvec[len] #]
solutions(ks : keystream) : PRED[MfClfsr] =
  { r : MfClfsr | ∀(i:below(len(ks))) : stream(r,i) = bits(ks)(i) }
```

The notation `[# .. #]` is used for labelled-product types. The length `len(ks)` of a keystream `ks:keystream` is thus even and bigger than 12, with a bit vector `bits(ks)` of this length.

For both the evenly and oddly numbered bits of keystream, we can look at the 20 bits of filter-function input that produce them. These will be described as even or odd "subsolutions", like in:

```
subsolutions_even(ks) : PRED[bvec[len(ks)/2 + 19]] =
  { s : bvec[len(ks)/2 + 19] | ∀(shiftnr:below(len(ks)/2)) :
      MfCfilfun20(λ(i:below(20)) : s(shiftnr + i))
        = bits(ks)(2*shiftnr) }
subsolutions_odd(ks) : PRED[bvec[len(ks)/2 + 19]] =
  { t : bvec[len(ks)/2 + 19] | ∀(shiftnr:below(len(ks)/2)) :
      MfCfilfun20(λ(i:below(20)) : t(shiftnr + i))
        = bits(ks)(2*shiftnr + 1) }
```

[4] Unless one has malicious intentions.

One sees that this formalisation makes the boundaries involved clearly visible.

In a next step we use the feedback function of the Mifare Classic LFSR to relate these two subsolutions. In order to do so we need to split the original feedback function, described in Section 3 as `MfCfeedback`, into two parts:

```
feedback_even(bv:bvec[24]) : bit =
  bv(0) XOR bv(5) XOR bv(6) XOR bv(7) XOR bv(12) XOR bv(21)
feedback_odd(bv:bvec[24]) : bit =
  bv(2) XOR bv(4) XOR bv(7) XOR bv(8) XOR bv(9) XOR bv(12) XOR
  bv(13) XOR bv(14) XOR bv(17) XOR bv(19) XOR bv(20) XOR bv(21)
```

in such a way that the original feedback is obtained as:

```
MfCfeedback(r) = (feedback_even(λ(i:below(24)) : r(2*i))
                  XOR feedback_odd(λ(i:below(24)) : r(2*i+1)))
```

The match that we seek between even and odd subsolutions is expressed by the following relation between two bit vectors.

```
shift2match?(ebv:bvec[25], obv:bvec[25]) : bool =
  feedback_even(λ(i:below(24)) : ebv(i))
      = (ebv(24) XOR feedback_odd(λ(i:below(24)) : obv(i)))
   ∧feedback_even(λ(i:below(24)) : obv(i))
        = (obv(24) XOR feedback_odd(λ(i:below(24)) : ebv(i+1)))
```

It is used to define matching subsolutions for a given keystream:

```
subsolutions(ks) =
  { (s : (subsolutions_even(ks)), t : (subsolutions_odd(ks))) |
     ∀(shiftnr:below(len(ks)/2 - 5)) :
       shift2match?(λ(i:below(25)) : s(i+shiftnr),
                    λ(i:below(25)) : t(i+shiftnr)) }
```

The main result then says:

```
∀(st : (subsolutions(ks)), shiftnr:below(len(ks))) :
  MfCfilfun(Advance(merge(ks)(st), shiftnr-9)) = bits(ks)(shiftnr)
```

where the `merge` function yields an LFSR state:

```
merge(ks)(st) : MfClfsr = λ(i:below(LfsrSize)) :
  IF even?(i) THEN proj_1(st)(i/2) ELSE proj_2(st)(i/2 - 1) ENDIF
```

The main result expresses a correctness property: the bits of a keystream can be obtained by applying the filter function to a merge of matching (even and odd) subsolutions. As sketched in [2, §§6.3], the set of such subsolutions can be calculated efficiently, from which a merged LFSR state results. The above correctness result shows that this process can be seen as an inverse to the filter function of the Mifare Classic card.

6.2 The odd-from-even attack

It is possible to improve upon the two-table attack when sufficiently many bits of the keystream are known. In the previous section we saw that subsolutions of even bits of the LFSR can be computed from the even bits of the keystream. Suppose we have an even subsolution of 48 bits. The following property specifies that a subsolution of 48 even bits corresponds to a given LFSR state.

```
StateAndEvens(r:MfClfsr, e:bvec[LfsrSize]) : bool
  = ∀(i:below(HalfLfsrSize)) :
      e(i) = r(2*i) ∧ e(HalfLfsrSize + i) = advance(r, LfsrSize)(2*i)
```

Given such a even subsolution, we can algebraically obtain the odd bits of the LFSR.

Consider as illustration the 4-bits LFSR with generating polynomial $x^4 + x^1 + 1$. If the bits of `advance(r, LfsrSize)` are named $r_4..r_7$, we can express these in terms of initial LFSR bits $r_0..r_3$:

$$
\begin{aligned}
r_4 &= r_0 \oplus r_3;\\
r_5 &= r_1 \oplus r_4 = r_0 \oplus r_1 \oplus r_3;\\
r_6 &= r_2 \oplus r_5 = r_0 \oplus r_1 \oplus r_2 \oplus r_3;\\
r_7 &= r_3 \oplus r_6 = r_0 \oplus r_1 \oplus r_2.
\end{aligned}
$$

For an even subsolution the even variables r_0, r_2, r_4 and r_6 are known. This leaves us with a system of four linear equations in four unknowns (the odd variables r_1, r_3, $r5$ and r_7). Solving the system gives

$$
\begin{aligned}
r_1 &= r_2 \oplus r_4 \oplus r_6;\\
r_3 &= r_0 \oplus r_4;\\
r_5 &= r_2 \oplus r_6;\\
r_7 &= r_0 \oplus r_4 \oplus r_6.
\end{aligned}
$$

In particular, we now have expressed the missing odd bits of the initial LFSR ($r1$ and $r3$) in terms of the bits of the even subsolution.

This computation can also be performed for the Mifare Classic LFSR. For example the first odd bit equals $e_1 \oplus e_2 \oplus e_6 \oplus e_{11} \oplus e_{13} \oplus e_{19} \oplus e_{21} \oplus e_{22} \oplus e_{23} \oplus e_{25} \oplus e_{28} \oplus e_{30} \oplus e_{32} \oplus e_{34} \oplus e_{36} \oplus e_{37} \oplus e_{38} \oplus e_{39} \oplus e_{41} \oplus e_{42}$. The equations for the value of the odd bits in terms of the even bits were obtained with an external program, but their correctness can readily be verified within PVS.

```
∀(r:MfClfsr, e:bvec[LfsrSize]) :
  StateAndEvens(r,e) IMPLIES (
    (r(1) = (e(1) XOR e(2) XOR e(6) XOR e(11) XOR e(13) XOR
             e(19) XOR e(21) XOR e(22) XOR e(23) XOR e(25) XOR
             e(28) XOR e(30) XOR e(32) XOR e(34) XOR e(36) XOR
             e(37) XOR e(38) XOR e(39) XOR e(41) XOR e(42)))
  AND
% similarly for r(3), r(5) .. r(47)
```

These observations lead to the following efficient attack. Suppose we have 58 bits of keystream, that is we have 29 bits of even and odd keystream each. Using the 29 bits of even keystream we do a depth-first search to find the even subsolutions of 48 bits using the extension method of Section 6.1. For each even subsolution we can compute the odd bits of the LFSR. These in turn determine the odd bits of the keystream, which can be matched against the observed odd keystream.

This attack is more efficient, because the odd subsolution is obtained directly from the even subsolutions (an $O(1)$ operation) whereas in the two-table attack each even subsolution has to be matched against each odd subsolution ($O(n^2)$ when done naively, $O(n \log(n))$ using a sorting operation on the feedback values).

Conclusions

We have described the basic logical details of the Mifare Classic card, focussing on its vulnerabilities and on two exploits. In the theorem prover PVS we have proved essential correctness results, while abstracting away from for instance matters of efficiency.

Many of the details of the formalisation are inherently specific to the Mifare Classic card and its weaknesses. However, this work does show that formalisations are relatively easy to do and can be both precise and readable. This makes them a solid base for the documentation and analysis of cryptographic systems. Thus, this paper suggests to card producers that they do such formalisations themselves, before bringing a card onto the market.

Acknowledgments

The reverse engineering and cryptanalysis of the Mifare Classic chip was very much a team effort and we thank the other members of the Mifare team for their hard work. Peter van Rossum in particular is acknowledged for his contributions to this paper.

References

1. The Boyer-Moore theorem prover. `www.computationallogic.com/software/nqthm/`.
2. F. Garcia, G. de Koning Gans, R. Muijrers, P. van Rossum, R. Verdult, R. Wichers Schreur, and B. Jacobs. Dismantling MIFARE Classic. In S. Jajodia and J. Lopez, editors, *Computer Security – ESORICS 2008*, number 5283 in Lect. Notes Comp. Sci., pages 97–114. Springer, Berlin, 2008.
3. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in Lect. Notes Comp. Sci. Springer, Berlin, 2002.
4. K. Nohl, D. Evans, Starbug, and H. Plötz. Reverse-engineering a cryptographic RFID tag. In *17th USENIX Security Symposium*, pages 185–194, San Jose, CA, USA, 2004.

5. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.
6. The Coq proof assistant. `http://coq.inria.fr`.
7. The Isabelle proof assistant. `http://isabelle.in.tum.de`.
8. W.G. Solomon. *Shift register sequences*. Aegean Park Press, Laguna Hills, Ca., 1982.
9. The PVS Specification and Verification System. `http://pvs.csl.sri.com`.
10. H.C.A. van Tilborg. *Fundamentals of Cryptology: a professional reference and interactive tutorial*. Kluwer Academic Publishers, 2000.