KeeLoq

Robert R. Enderlein

School of Computer and Communication Sciences

Semester Project

January 2010

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Pouyan Sepehrdad
EPFL / LASEC

Contents

Introduction

KeeLoq is a proprietary block cipher owned by Microchip, and is used in remote key-less entry systems from several car manufacturers — such as Chrysler, Daewoo, Fiat, GM, Honda, Toyota, Volvo, VW, Jaguar, etc. [19] — as well as for garage door openers. After the confidential specifications have been leaked on a Russian website [12] in 2006, several cryptanalysts have found substantial weaknesses in the design of the algorithm [2, 5, 9] and the hardware on which it is implemented [7, 10].

The objectives of this semester project are to understand and implement the attacks against KeeLoq, try to improve them, and possibly to test them in real life. Side channel attacks and weaknesses of the physical implementations are out-of-scope.

The report starts with a literature review of KeeLoq. In chapter 1, the KeeLoq cipher, key derivation functions, and authentication protocols are described. The state-of-the-art attacks against KeeLoq are summarized in chapters 2 (attacks against the cipher) and 3 (attacks against the protocols).

Our contributions are presented in the subsequent chapters. We have captured and analyzed radio traces of several cars and garage keys which supposedly use KeeLoq (chapter 4). We have implemented two key-recovery attacks in software (chapter 5). Our efforts to improve the state of the art in KeeLoq cryptanalysis are documented in chapter 6, however, no positive results have been achieved. Finally, we propose a slight modification of the KeeLoq key schedule which renders the cipher immune against all published cryptanalysis (chapter 7).

Chapter 1

Description of KeeLoq

The purpose of this chapter is to present the KeeLoq encryption and decryption algorithms, the diverse key derivation functions used by KeeLoq, and the authentication protocols in which KeeLoq is used.

1.1 Encryption

KeeLoq is a block cipher with a 64-bit key and a 32-bit block size [2, 9, 12]. The cipher operates on two registers:

- A 64-bit key register;
- A 32-bit text non-linear feedback shift register (NLFSR).

The key register

The 64-bit key register operates as a simple circular-shift register. The register is initially filled with the encryption key $k_{63..0}$. Let the state of the key register in round i be denoted by:

$$K_{63..0}^{\langle i \rangle} = \left(K_{63}^{\langle i \rangle}, K_{62}^{\langle i \rangle}, \ldots, K_{0}^{\langle i \rangle}\right) \tag{1.1}$$

$$\text{with} \quad K_{63..0}^{\langle 0 \rangle} = k_{63..0} \tag{1.2}$$

In each round, the contents of the register are simply rotated:

$$K_{63..0}^{\langle i+1 \rangle} = \mathrm{ROR}_1(K_{63..0}^{\langle i \rangle}) = \left(K_{0}^{\langle i \rangle}, K_{63}^{\langle i \rangle}, K_{62}^{\langle i \rangle}, \ldots, K_{1}^{\langle i \rangle}\right) \tag{1.3}$$

Where ROR_r is a circular rotation of r bits to the right.

The text register

The 32-bit text register operates as a non-linear feedback shift register. Let the state of the text register in round i be denoted by:

$$T_{31..0}^{\langle i \rangle} = \left(T_{31}^{\langle i \rangle}, T_{30}^{\langle i \rangle}, \ldots, T_{0}^{\langle i \rangle}\right) \tag{1.4}$$

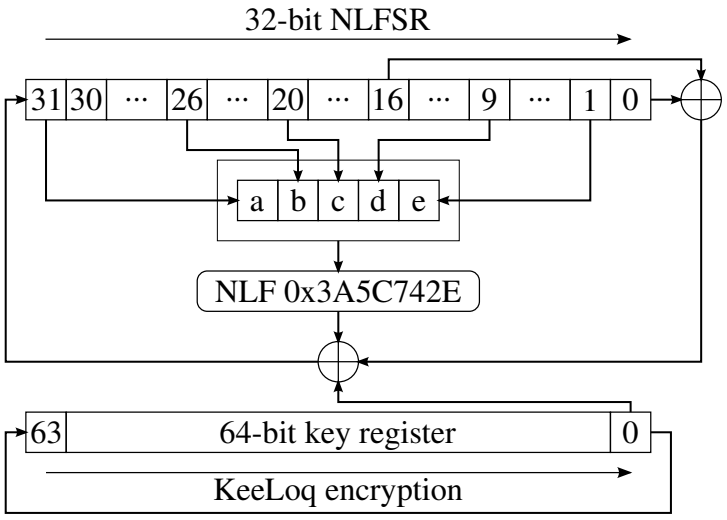


Figure 1.1: KeeLoq encryption algorithm [19]

The register is initially filled with the plaintext $P_{31..0}$:

$$T_{31..0}^{\langle 0 \rangle} = P_{31..0} \tag{1.5}$$

The ciphertext $C_{31..0}$ is read from the text register after 528 rounds:

$$C_{31..0} = T_{31..0}^{\langle 528 \rangle} \tag{1.6}$$

During each round, the 31 most significant bits (MSB) are shifted right by one position (becoming the 31 least significant bits (LSB) of the next round), and the MSB of the new round is computed from a non-linear combination of bits from the text and key register:

$$T_{31..0}^{\langle i+1 \rangle} = \left(\varphi^{\langle i \rangle}, T_{31}^{\langle i \rangle}, T_{30}^{\langle i \rangle}, \ldots, T_{1}^{\langle i \rangle}\right) \tag{1.7}$$

$$\varphi^{\langle i \rangle} = \mathrm{NLF}(T_{31}^{\langle i \rangle}, T_{26}^{\langle i \rangle}, T_{20}^{\langle i \rangle}, T_{9}^{\langle i \rangle}, T_{1}^{\langle i \rangle}) \oplus T_{16}^{\langle i \rangle} \oplus T_{0}^{\langle i \rangle} \oplus K_{0}^{\langle i \rangle} \tag{1.8}$$

Since there are only 32 possible inputs to the non-linear function NLF, it is usually defined by a lookup table. If we consider the input as a 5-bit number $i_{4..0} = \left(a, b, c, d, e\right)$ then $\mathrm{NLF}(i)$ is equal to the i^{th} LSB of the hexadecimal number $\texttt{3A5C742E}_{16}$. Alternatively, it is possible to write the NLF in algebraic form [2, 5, 9]:

$$\begin{aligned} \mathrm{NLF}(a, b, c, d, e) = {} & d \oplus e \oplus ac \oplus ae \oplus bc \oplus be \oplus cd \oplus de \\ & \oplus ade \oplus ace \oplus abd \oplus abc \end{aligned} \tag{1.9}$$

Figure 1.1 provides a more visual representation of the encryption processes.

1.1.1 Compact description

All bits except the LSB are just shifted in a given round, i.e.:

$$\forall k, 0 \leq k \leq 31, k \leq i + j \qquad T_j^{\langle i \rangle} = T_k^{\langle i+j-k \rangle} \tag{1.10}$$

It is therefore unnecessary to keep track of the state of the whole text register in each round: it is sufficient to keep track of the new bit introduced in each round, the MSB. The 32 initial bits of the text register and the MSB of the text register after each of the 528 rounds define a bitstream $L_{559..0}$ [5], which will be called *encryption state*.
The 32 LSB of the encryption state are the plaintext:

$$L_{31..0} = P_{31..0} \tag{1.11}$$

The bit $(31 + i)$ of the encryption state is the MSB of the text register after round i.

It is possible to simplify the description of the encryption process by observing that, during round $i+1$, only the bit L_{32+i} needs to be computed (from the bits $L_{31+i..i}$ and the key bit $k_0^{\langle i \rangle} = k_{i \bmod 64}$). The bits $L_{31+i..i}$ will be identical with the contents of the text register in round i. Formally:

$$\begin{aligned} &\forall i, 32 \leq i \leq 559 \\ &L_i = k_{i-32 \bmod 64} \oplus L_{i-32} \oplus L_{i-16} \oplus \mathrm{NLF}\left(L_{i-1}, L_{i-6}, L_{i-12}, L_{i-23}, L_{i-31}\right) \end{aligned} \tag{1.12}$$

The ciphertext is then read from the 32 MSB of the encryption state after 528 rounds:

$$C_{31..0} = L_{559..528} \tag{1.13}$$

1.1.2 Notation

In this report the following useful functions will be used:

$E_{k_{63..0}}$	Full KeeLoq encryption (528 rounds $= 8 \cdot 64 + 16$)
$f_{k_{63..0}}$	64 rounds of KeeLoq
$g_{k_{15+z..z}}$	16 rounds of KeeLoq $(0 \leq z \leq 48)$

Due to the very simple key schedule, a full KeeLoq encryption is just applying the 64-round-reduced encryption (f_k) $8\frac{1}{4}$ times with the same key k:

$$E_{k_{63..0}} = g_{k_{15..0}} \circ f^{(8)}_{k_{63..0}} \tag{1.14}$$

$$\text{and} \qquad f_{k_{63..0}} = g_{k_{63..48}} \circ g_{k_{47..32}} \circ g_{k_{31..16}} \circ g_{k_{15..0}} \tag{1.15}$$

1.2 Decryption

All steps in the encryption process can be reversed: the only bit that was "dropped" from the text register when performing round i ($T_0^{\langle i \rangle}$) can be recomputed from the bit that was "added" ($T_{31}^{\langle i+1 \rangle}$) and the bits that were just shifted ($T_{30..0}^{\langle i+1 \rangle} = T_{31..1}^{\langle i \rangle}$) [2, 9, 12]:

$$\begin{aligned} T_{31}^{\langle i+1 \rangle} &= \mathrm{NLF}(T_{31}^{\langle i \rangle}, T_{26}^{\langle i \rangle}, T_{20}^{\langle i \rangle}, T_{9}^{\langle i \rangle}, T_{1}^{\langle i \rangle}) \oplus T_{16}^{\langle i \rangle} \oplus T_{0}^{\langle i \rangle} \oplus K_{0}^{\langle i \rangle} \\ T_{0}^{\langle i \rangle} &= \mathrm{NLF}(T_{31}^{\langle i \rangle}, T_{26}^{\langle i \rangle}, T_{20}^{\langle i \rangle}, T_{9}^{\langle i \rangle}, T_{1}^{\langle i \rangle}) \oplus T_{16}^{\langle i \rangle} \oplus T_{31}^{\langle i+1 \rangle} \oplus K_{0}^{\langle i \rangle} \\ &= \mathrm{NLF}(T_{30}^{\langle i+1 \rangle}, T_{25}^{\langle i+1 \rangle}, T_{19}^{\langle i+1 \rangle}, T_{8}^{\langle i+1 \rangle}, T_{0}^{\langle i+1 \rangle}) \oplus T_{15}^{\langle i+1 \rangle} \oplus T_{31}^{\langle i+1 \rangle} \oplus K_{0}^{\langle i \rangle} \end{aligned} \tag{1.16}$$

Figure 1.2 provides a more visual representation of the decryption processes.

1.2.1 Compact description

The 32 MSB of the encryption state are the ciphertext:

$$L_{559..528} = C_{31..0} \tag{1.17}$$

The decryption process is then very similar to the encryption process (starting at round 527 down to 0):

$$\begin{aligned} &\forall i, 527 \geq i \geq 0 \\ &L_i = k_{i \bmod 64} \oplus L_{i+32} \oplus L_{i+16} \oplus \mathrm{NLF}\left(L_{i+31}, L_{i+26}, L_{i+20}, L_{i+9}, L_{i+1}\right) \end{aligned} \tag{1.18}$$

We can then read the plaintext from the 32 LSB of the encryption state register after 528 rounds:

$$P_{31..0} = L_{31..0} \tag{1.19}$$

1.3 Key derivation functions

There are several key derivation functions [12]. The keys of the individual devices k_{dev} are derived from a more or less random seed (which may include bits from the publicly known device serial number $S_{27..0}$ and some padding) and the manufacturer key k_{man}. The latter is constant over a wide range of products by the same manufacturer [2], for example all cars of a given model in a given year.

There are two methods to derive the device key: either by performing an XOR between the manufacturer key and the seed (see Figure 1.3(a)), or by decrypting the first and second half of the seed with the manufacturer key using the KeeLoq algorithm (Figure 1.3(b)) [12].

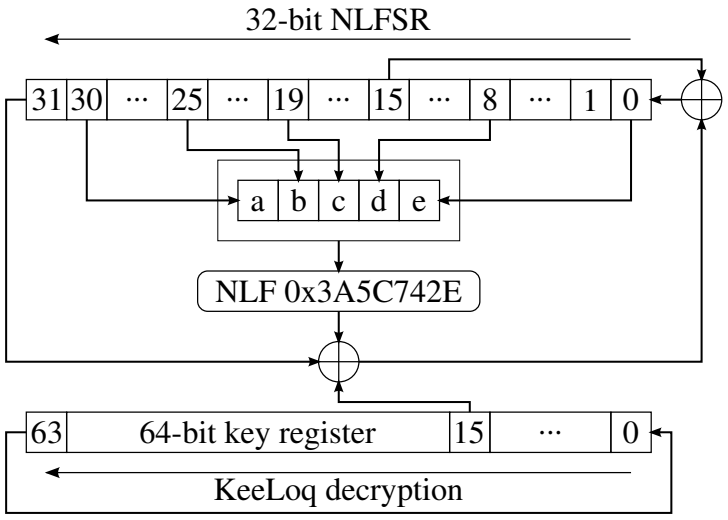


Figure 1.2: KeeLoq decryption algorithm [19]

The seed may contain 0, 32, 48 or 60 random bits[1]. The seed is generated as shown in Table 1.1.

Random bits	Seed 2	Seed 1
0	$0100_2 \parallel S_{27..0}$	$0010_2 \parallel S_{27..0}$
32	$0000_2 \parallel S_{27..0}$	32 random bits
48	$0000_2 \parallel S_{27..16} \parallel$ 16 random bits	32 random bits
60	$0000_2 \parallel$ 28 random bits	32 random bits

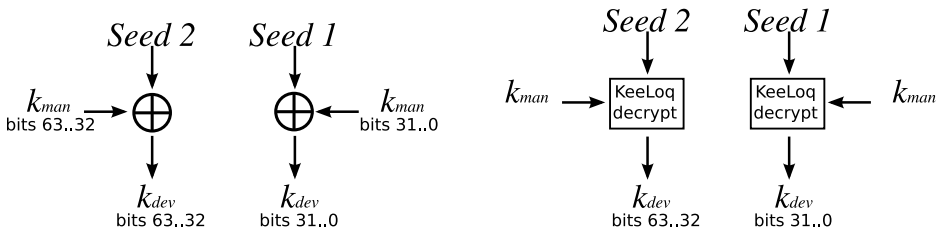
Table 1.1: Generation of the seed for the key derivation function [2, 11]

The manufacturer key k_{man} is a very interesting target for the attacker. If it is known, the security of the device key k_{dev} drops from 64 bits to 60, 48, 32 or even 0 bits.

1.3.1 Storage of the seed

The seed is usually stored in the encoder (remote control) when it is programmed, and must be paired with the decoder (the car or the garage) before use. To that effect, the user must make a special manipulation on the decoder module (for instance press the "learn" button) so that it enters the "learn" state [12]. The user must then press a special button or a button combination on the encoder so that it transmits its seed and serial number in plaintext to the decoder.

[1]Some models of encoders may not support all variants: HCS101 and HCS365 only support the 0 random bit variant; HCS2xx, 30x and 320 support up to 32 random bits; HCS360 and HCS361 support up to 48 random bits; HCS362 and 4xx support up to 60 bits.



(a) XOR-based (b) Decryption-based

Figure 1.3: Key derivation functions [2, 11, 12]

It is up to the car manufacturer to decide if to inhibit the transmission of the seed in the encoder after a certain number of button presses [11]. This is a compromise between security and user-friendliness since the user will not be able to re-associate the key with another decoder (or even the same decoder if the latter erases the association parameters).

If the manufacturer decides not to inhibit seed transmission, an attacker who has access to the encoder for a few seconds can retrieve the seed (if the corresponding k_{man} is known, she will then be able to compute k_{dev} immediately).

The most user-friendly and least secure design is of course not to use a seed at all, since the manufacturer can ship the encoder with one button less, and the user will not have to do a complicated manipulation with the encoder during the learning phase. It is worth mentioning that all the integrated circuits (IC) analyzed by Eisenbarth *et al.* [7] used a seed derived directly from the device serial number (0 random bits).

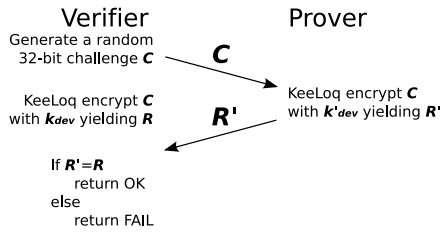
1.4 Authentication protocols

KeeLoq is used chiefly for authentication. There are two different protocols that are used to that effect.

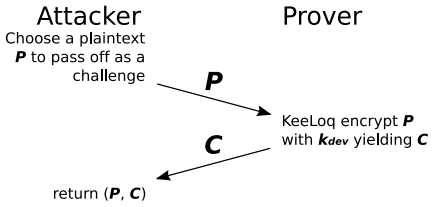
1.4.1 Identify Friend-or-Foe (IFF)

The IFF protocol is a challenge-response protocol implemented in modules supporting two-way communication. Microchip's HCS410 module for instance can be used in an RFID-like proximity-based access control. Microchip also suggests using this mode to verify the compatibility of hardware in modular systems [14].

The IFF protocol is extremely simple [14]. The verifier can request the serial number of the prover, or some other configuration options, request data to be written in the prover module or perform a challenge.



(a) Regular IFF [14]



(b) The attacker spoofs the verifier to get the ciphertext corresponding to a chosen plaintext.

Figure 1.4: Identify-friend-or-foe

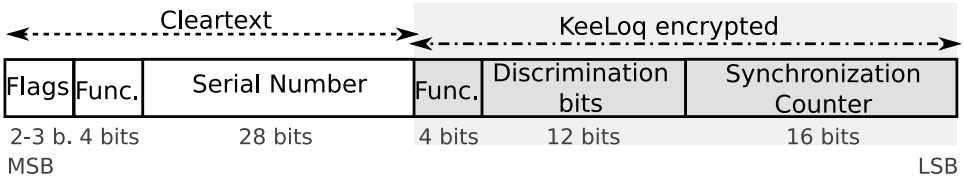


Figure 1.5: Hopping code message format [13]

For the latter, the verifier is expected to know the key k_{dev} of the prover. The verifier sends a 32-bit challenge to the prover, which the latter then encrypts and sends back (see Figure 1.4(a)).

It is worth mentioning that the verifier never needs to prove its identity: by spoofing the verifier, the attacker can use the prover as an encryption oracle (Figure 1.4(b)).

1.4.2 Hopping codes

The hopping code protocol is implemented in devices which are capable of one-way communication only. Microchip produces several modules which are capable of sending hopping codes[2], and several devices capable of decoding them again[3] (there are no modules which combine encryption and decryption). The typical encoders are car keys and garage remotes, and the decoders are the car or the garage door. The advantage of this solution is that the encoders (which are typically limited in power) never have to listen to the radio channel.

The protocol consists of only one 66/67-bit message transmitted LSB first from the encoder to the decoder (typically over a radio channel) [13]. The contents of the message is detailed in Figure 1.5.

[2]HCS 101, 200, 201, 300, 301, 320, 360, 361, 362, 365, 410, 412, 473.
[3]HCS 500, 512, 515, and PIC 8-bit micro-controllers.

- The **synchronization counter** is a 16-bit number which is incremented with each message send by the encoder and is used for replay protection. The way the decoder handles this number is described below.

- The **discrimination value** is an arbitrary 12-bit number which is more or less unique for each encoder. In some integrated circuits (IC)[4] this number can be programmed, others just put the 10 LSB of the serial number in the 10 LSB of the discrimination value and pad the rest with zeros. Some ICs may use up to 2 bits (the 2 MSB) of this discrimination value to increase the effective length of the synchronization counter to 18 bits[5].

- The **function bits** are application dependent. This allows up to 16 different functions (such as lock the car, open the car, open the trunk, etc.) to be performed. On most models, one combination is reserved for the "learning" mode[6].

- The **serial number** is a unique identifier of the encoder.

- The **flag bits** vary from IC model to model: there might be a bit for "transmission repetition", one for "battery low"[7], or one or several CRC bits (computed over the whole transmission).

The synchronization counter, discrimination value and function bits are encrypted with KeeLoq keyed with k_{dev}. The serial number, a copy of the function bits, and the flag bits are sent in cleartext.

Synchronization between encoder and decoder

Since the encoder can transmit while being out-of-range of the decoder, the counter value at the decoder might become desynchronized with that of the encoder (for example, if the user puts his car key in his pocket and a buttons get inadvertently and repeatedly pressed, or if a child plays with the car key). To cope with these eventualities, the decoder defines three ranges of counter values (Figure 1.6). Let R be the currently received counter value at the decoder, and L be the counter value stored in the decoder; then the three ranges are as follows:

[4] HCS300 and HCS301.

[5] In models supporting this option, for compatibility reasons, the 2 MSB of the discrimination value can never be set to a non-zero value again once they both become zero [12]. This means that the counter first goes though 2^{18} values, but then continues on a cycle of period 2^{16}.

[6] In the learning mode, the 32 LSB bits of Figure 1.5 are replaced by the 32-bit seed transmitted in cleartext.

[7] This allows the decoder to warn the user.

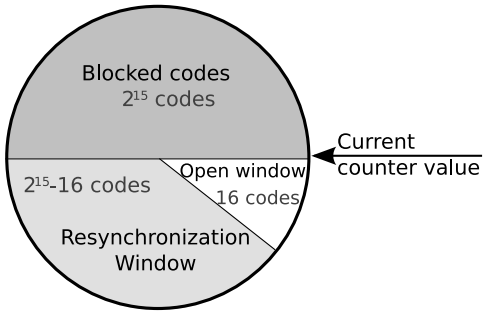


Figure 1.6: Windows in the KeeLoq hopping code [12]

- The **open window**: the received counter value is within 16 units [12] of the last value ($1 + L \leq R \leq 16 + L \pmod{2^{16}}$). In this case, the decoder accepts the transmission, performs the requisite function and updates its own synchronization counter $L \leftarrow R$.
- The **re-synchronization window**: the received counter value is within 2^{15} units [12] of the last value ($16 + L < R \leq 2^{15} + L \pmod{2^{16}}$). In that case, the decoder stores the value R and waits for the value $R + 1$. If the next message has a counter value of $R + 1$, the decoder performs the requisite function and updates its own synchronization counter $L \leftarrow R + 1$. This design exploits the natural reflex of the user to press the button on the encoder again if the expected function did not happen.
- The **blocked window**: the received counter value does not fall in either of the above two windows ($R \leq L$ or $R > 2^{15} + L$). In this case the decoder rejects the transmission and does not perform anything. This mechanism enforces replay protection (assuming the counter has not rolled over, which should not happen in the lifetime of a typical encoder).

Chapter 2

State-of-the-art attacks against KeeLoq

This chapter will start with a comparison of several published key-recovery attacks against the KeeLoq block cipher and widely spread hardware implementations. These attacks will then be described in varying levels of detail. Finally, work done on side channel attacks against hardware implementations will be briefly mentioned.

2.1 Comparison of key-recovery attacks

Table 2.1 provides a compact comparison of the various attacks against KeeLoq. A more detailed description of all attacks will be made in the following sections.

Attack	Requirements[a]	Time[b]	Success[c]	Source
Brute force	2 KP	2^{64}	100%	
Slide-algebraic	2^{16} KP	2^{53}	63%	[5]
Slide-meet-in-the middle	2^{16} KP	2^{45}	63%	[9]
Slide-correlation	2^{32} KP	$2^{50.6}$	91%	[2]
Slide-and-determine	2^{32} KP	$2^{31.1}$	63%[d]	[5]
Differential power analysis	≈30 PT	small	100%	[7]
Simple power analysis	1 PT	small	100%	[10]

[a] KP: Known plaintext-ciphertext pairs, PT: power traces.

[b] Running time in full KeeLoq encryptions, as estimated by the respective authors.

[c] Success probability, with the parameters suggested by the respective authors.

[d] This attacks works only against 63% of all *keys*.

Table 2.1: Comparison of different attacks against KeeLoq

2.2 Known-plaintext key-recovery attacks

The papers [2], [5], and [9] present different key recovery attacks against KeeLoq. They all exploit the simplistic key schedule of the cipher to mount a sliding attack.

2.2.1 Observations

In this subsection, two properties of the KeeLoq cipher that are exploited in all the key-recovery attacks will be described.

Directly determining key bits

Because the key is simply XORed to each bit of the encryption state L, it is possible to directly deduce some key bits if more than 32 consecutive bits of the encryption state are known [5]:
Knowing $L_{i..j}$ with $j > 32 + i$, one can directly compute:

$$k_{z \bmod 64} \quad \forall z, i \leq z \leq j - 32$$

by observing that

$$k_{z \bmod 64} = L_z \oplus L_{z+32} \oplus L_{z+16} \oplus \mathrm{NLF}\left(L_{z+31}, L_{z+26}, L_{z+20}, L_{z+9}, L_{z+1}\right) \tag{2.1}$$

Slid pairs

Two plaintexts $P^{[i]}$ and $P^{[j]}$ form a *slid pair* for a given key k iff the latter can be obtained by encrypting the former by only 64 rounds [2, 9], i.e.:

$$P^{[j]} = f_k(P^{[i]}) \tag{2.2}$$

After 64 rounds, each bit of the key k will have been mixed exactly once with the plaintext. Slid pairs are very attractive from a cryptanalysis point of view, and indeed most of the published attacks make use of this concept: if the cryptanalyst can find one slid pair, he can effectively reduce the cryptanalysis to 64 rounds.

2.2.2 Slide-correlation attack

This attack was presented in the paper by Bogdanov [2]. It requires the full codebook (2^{32} plaintext-ciphertext pairs, which requires 16 GiB of memory) and has a success probability of about 91% for the proposed parameters[1]. The running time was estimated to be about $2^{50.6}$ KeeLoq encryptions[2].

The idea behind the attack is based on the following two observations:

[1] This probability can be made larger or smaller at the expense of the running time by generating more slid pairs.

[2] Assuming that one memory access costs as much as performing 4 rounds of KeeLoq.

- With one slid pair $O^{[i]} = f_k(I^{[i]})$, and the first 16 bits of the key $k_{15..0}$, it is possible to quickly generate another slid pair as follows:

$$I^{[i+1]} = g^{(-1)}_{k_{15..0}}(E(I^{[i]})) \tag{2.3}$$
$$O^{[i+1]} = g^{(-1)}_{k_{15..0}}(E(O^{[i]})) \tag{2.4}$$

  where $E(x)$ is looked up in the codebook. Indeed:

$$I^{[i+1]} = g^{(-1)}_{k_{15..0}}(g_{k_{15..0}}(f^{(8)}_k(I^{[i]}))) \ = f^{(8)}_k(I^{[i]}) \tag{2.5}$$
$$O^{[i+1]} = g^{(-1)}_{k_{15..0}}(g_{k_{15..0}}(f^{(8)}_k(O^{[i]}))) = f^{(8)}_k(O^{[i]}) = f^{(9)}_k(I^{[i]}) = f_k(I^{[i+1]}) \tag{2.6}$$

- The non-linear function is biased. Indeed, for uniformly distributed random variables (a, b, c, d, e):

$$\Pr\{\mathrm{NLF}(a, b, c, d, e) = d \oplus e\} = \frac{1}{2} + \frac{1}{8} \tag{2.7}$$

  Given enough slid pairs (about 2^8), one can exploit this property to quickly guess 32 more key bits one-by-one with a very high probability. The last 16 bits of the key can then be directly computed.

In the attack, the 16 LSB of the key $k_{15..0}$ have to be guessed[3] (time 2^{16}), and given an initial $I^{[0]}$ plaintext, a possible slid pair $O^{[0]}$ (32 bits) has to be guessed (time 2^{32}). By exploiting the two observations above, it is possible to compute (and try out) a candidate key faster than 2^{16} full encryptions.

2.2.3 Slide-and-determine attack

The slide and determine attack by Courtois *et al.* [5, Section 5.2] is an improvement over the attack by Bogdanov. It also requires the full codebook (2^{32} plaintext-ciphertext pairs), but requires only $2^{31.1}$ KeeLoq encryptions on average (assuming we get the codebook for free). Another property of this attack is that it works only against about 63% of all keys, so this attack can be made harmless by avoiding weak keys.

The main idea of this attack is to find a plaintext $P^{[i]}$ that is a fixed point with respect to a 64-round-reduced KeeLoq, i.e., $f_k(P^{[i]}) = P^{[i]}$. Such a fixed point exists for about 63% of all keys. The attack operates in two stages:

- All fixed points of $f^{(8)}$ (512 rounds of KeeLoq) are found. All fixed points of f will necessarily be fixed points of $f^{(8)}$. Since the codebook only allows $E = g \circ f^{(8)}$ to be computed, only 16 of the 32 bits of $f^{(8)}$

[3]The cryptanalyst does not really guess the bits, but he iterates over all possible values.

can be determined (since KeeLoq only replaces one bit per round), but that is sufficient to filter out a large number of wrong guesses. After this step, around 2^{16} candidates for a fixed point are left.

For each of these candidates, it is assumed that they are indeed a fixed point of $f^{(8)}$, and the corresponding first 16 bits of the key $k^{[i]}_{15..0}$ are directly computed. The triplet $\langle P^{[i]}, C^{[i]}, k^{[i]}_{15..0} \rangle$ is then stored.

- It is now assumed that for each of the stored triplets, $P^{[i]}$ is a fixed point of f. For each of the stored triplets, the next 16 bits of the key $k^{[i]}_{31..16}$ are guessed. Using that, it is now possible to directly compute the remaining key bits. With some trial encryptions with the known plaintext-ciphertext pairs, the candidate key is accepted or rejected.

2.2.4 Slide-algebraic attack

In the same paper, Courtois *et al.* [5, Section 7.2] presented their findings on applying algebraic attacks on the KeeLoq cipher. In their experiments they could not mount a direct algebraic attack on the full cipher (after 128 rounds the performance was becoming unacceptably slow). For 64 rounds however, they got very good results: using the SAT-solver MiniSat, the key could be recovered in just 2.3 seconds (2^{32} CPU clocks) on average.

As the same suggests, this attack combines a slide attack with their algebraic attack. By the birthday paradox, with 2^{16} plaintext-ciphertext pairs it is very probable (about 63%) that there is at least one slid pair $f_k(P^{[i]}) = P^{[j]}$, yielding the first 64 equations. 64 additional equations can be obtained by observing that the corresponding ciphertexts of the slid-pair $C^{[i]} = E(P^{[i]})$ and $C^{[j]} = E(P^{[j]})$ also form a slid pair but for a key that is shifted:

$$\begin{aligned} f_k(P^{[i]}) &= P^{[j]} \\ g_{k_{15..0}}(f_k^{(9)}(P^{[i]})) &= g_{k_{15..0}}(f_k^{(8)}(P^{[j]})) \\ g_{k_{15..0}}(g_{k_{63..48}}(g_{k_{47..32}}(g_{k_{31..16}}(g_{k_{15..0}}(f_k^{(8)}(P^{[i]})))))) &= g_{k_{15..0}}(f_k^{(8)}(P^{[j]})) \\ g_{k_{15..0}}(g_{k_{63..48}}(g_{k_{47..32}}(g_{k_{31..16}}(C^{[i]})))) &= C^{[j]} \\ f_{k_{15..0,63..16}}(C^{[i]}) &= C^{[j]} \end{aligned} \tag{2.8}$$

To summarize, these are the equations that are used:

$$\begin{aligned} P^{[j]} &= f_k(P^{[i]}) && \text{(64 equations)} \\ C^{[j]} &= f_{k_{15..0,63..16}}(C^{[i]}) && \text{(64 equations)} \end{aligned} \tag{2.9}$$

These 128 equations are transformed to Conjunctive Normal Form with a ANF-to-CNF converter (see Section 5.2.3), which are then given to the SAT-solver. If the given pair was not a slid pair, the SAT-solver will either output

garbage that can be quickly filtered out by performing some trial encryptions on the other known plaintexts, or will output "unsatisfiable". If however, the pair was indeed a slid pair, then the SAT-solver will output the key[4].

Performing this attacks requires about 2^{64} CPU clocks, which corresponds to 2^{53} KeeLoq encryptions.

We have implemented this attack (see Section 5.2.2).

2.2.5 Slide-meet-in-the-middle attack

This attack was first described by Indesteege *et al.* [9], and requires 2^{16} plaintext-ciphertext pairs. Like the previous attack, it is based on the hope to find a slid pair $f_k(P^{[i]}) = P^{[j]}$, which happens with probability 63%[5]. However, this attack gives an explicit algorithm to find the key given a potential slid pair, and does not require a black-box SAT-solver.

This algorithm uses a clever way of fixing some bits of the encryption state and the key to avoid unnecessary computation. It uses a hash table to quickly decimate the number of candidate slid pairs to consider. The algorithm runs faster than the slide-algebraic attack: 2^{54} register clock cycles, equivalent to about 2^{45} full KeeLoq encryptions.

Description of the algorithm

1. The first step is to guess the first 16 bits of the key $k_{15..0}$. With this information, all plaintexts $P^{[i]}$ are partially encrypted by 16 rounds (yielding $L^{[i]}_{47..0}$) and all ciphertexts $C^{[j]}$ are partially decrypted by 16 rounds (yielding $L^{[j]}_{623..576}$).

2. Next, 16 bits denoted by $\alpha_{15..0}$ are guessed, and $L^{[j]}_{63..48}$ is set to α. The key bits $k^{[j]}_{63..48}$ are computed from $L^{[j]}_{95..48} = P^{[j]}||\alpha$. With this information, all ciphertexts are partially decrypted by 16 more rounds (yielding $L^{[j]}_{623..560}$). The tuples $\langle L^{[j]}_{79..48}, L^{[j]}_{591..560}, k^{[j]}_{63..48}\rangle$ are then saved in a hash table indexed by $L^{[j]}_{575..560}$.

3. $L^{[i]}_{63..48}$ is now set to α, in effect forcing all pairs $\langle P^{[i]}, P^{[j]}\rangle$ to be slid plaintexts for some key $k^{[i,j]}$ (where $k^{[i,j]}_{63..48} = k^{[j]}_{63..48}$ and $k^{[i,j]}_{31..0} = k^{[i]}_{31..0}$). For each plaintext $P^{[i]}$, the key bits $k^{[i]}_{31..16}$ are computed from $L^{[i]}_{63..16}$. Using this information, all ciphertexts $C^{[i]}$ are encrypted by 16 rounds (yielding $L^{[i]}_{575..528}$).

[4] We believe that this algorithm, as given in [5, Section 7.2], is mistaken/incomplete, and will have a much lower probability of success than claimed, since it is possible that the system of equations has several solutions, so the SAT-solver might miss the correct one. See Section 5.2.2.

[5] Again, this probability can be increased at the cost of worse running time by having more plaintext-ciphertext pairs available.

for all possible values of $k_{15..0}$ **do**
  for all plaintexts $P^{[i]}$, $0 \leq i < 2^{16}$ **do**
    Partially encrypt $P^{[i]}$ 16 rounds, yielding $L^{[i]}_{47..0}$.
    Partially decrypt $C^{[j]}$ 16 rounds, yielding $L^{[j]}_{623..576}$.
  for all possible values of $\alpha_{15..0}$ **do**
    for all plaintexts $P^{[j]}$, $0 \leq i < 2^{16}$ **do**
      $L^{[j]}_{63..48} \leftarrow \alpha_{15..0}$.
      Determine the key bits $k^{[j]}_{63..48}$ from $P^{[j]}||\alpha$.
      Partially decrypt the ciphertexts $C^{[j]}$, yielding $L^{[j]}_{623..560}$.
      Insert $\langle L^{[j]}_{79..48}, L^{[j]}_{591..560}, k^{[j]}_{63..48}\rangle$ in a hash table indexed by $L^{[j]}_{575..560}$.
    for all plaintexts $P^{[i]}$, $0 \leq i < 2^{16}$ **do**
      $L^{[i]}_{63..48} \leftarrow \alpha_{15..0}$.
      Determine the key bits $k^{[i]}_{31..16}$ from $\alpha||L^{[i]}_{47..16}$.
      Partially encrypt the ciphertext $C^{[i]}$, yielding $L^{[i]}_{575..528}$.
      for all collisions $L^{[i]}_{575..560} = L^{[j]}_{575..560}$ in the hash table **do**
        Determine the key bits $k^{[i,j]}_{47..32}$ from $L^{[j]}_{79..48}||L^{[i]}_{47..32}$.
        Determine the key bits $k'^{[i,j]}_{47..32}$ from $L^{[j]}_{591..560}||L^{[i]}_{559..544}$.
        if $k^{[i,j]}_{47..32} = k'^{[i,j]}_{47..32}$ **then**
          $k^{[i,j]} \leftarrow k^{[j]}_{63..48}||k^{[i,j]}_{47..32}||k^{[i]}_{31..0}$
          Encrypt several known plaintexts with the key $k^{[i,j]}$.
          if the correct ciphertexts were found **then**
            return success (the key is $k^{[i,j]}$)
return failure (there was no slid pair)

Figure 2.1: The attack algorithm in pseudocode [9, Fig.4]

4. All pairs $\langle P^{[i]}, P^{[j]}\rangle$ which satisfy $L^{[i]}_{575..560} = L^{[j]}_{575..560}$ are looked up in the hash table: the corresponding ciphertexts $\langle C^{[i]}, C^{[j]}\rangle$ are then slid ciphertexts for some key $k'^{[i,j]}$.

5. Subsequently, only the pairs retrieved from the hash table that also satisfy $k'^{[i,j]} = k^{[i,j]}$ are kept (only the bits 47 to 32 need to be checked, since the others are always the same); we can compute the two keys from $P^{[j]}||L^{[i]}_{63..32}$ and $L^{[j]}_{591..560}||L^{[i]}_{559..544}$. For pairs $\langle i, j\rangle$ which are not slid pairs, this equality holds with probability 2^{-16}.

6. Finally, the key $k'^{[i,j]}$ of all remaining pairs is used to make some trial encryptions with some known plaintext-ciphertext pairs. Only the correct key will remain after this step.

Figure 2.1 shows a pseudocode of the attack algorithm.

Running time analysis

Let R be the cost of one round of KeeLoq (encryption, decryption or directly computing a key bit). Since we need only just slightly more than 2 MiB of memory, we can assume that everything is located in the cache of the CPU and that memory access is therefore free. Let N_{col} be the number of collisions in the hash table and let V be the cost of verifying one collision.

- 2^{16} iterations of the outer loop.
  - 2^{16} iterations in the next loop.
    - 16 rounds of encryption and 16 rounds of decryption are performed.
  - 2^{16} iterations in the loop where $\alpha_{15..0}$ is guessed.
    - 16 bits of the key are determined and 16 rounds of decryption are performed per iteration in the next loop.
    - 2^{16} iterations in the loop over all plaintexts.
      - 16 bits of the key are determined and 16 rounds of encryption are performed.
      - All tuples can be fetched from hash table in constant time. There is an average of N_{col} results.
        - For each collision, some work needs to be done: V.

The total cost is therefore:

$$2^{16} \cdot (2^{16} \cdot (2 \cdot 16 \cdot R) + 2^{16} \cdot (2 \cdot 16 \cdot R + 2^{16} \cdot (2 \cdot 16 \cdot R + N_{col} \cdot V))) \quad (2.10)$$

With 2^{16} known plaintext-ciphertext pairs, 2^{16} elements will be inserted in a hash table indexed by a 16-bit key. The expected number of elements fetched is therefore $N_{col} = 1$.

As for verifying that the two computed keys $k_{47..32}^{[i,j]}$ and $k_{47..32}^{\prime[i,j]}$ are identical, it can be done bit-per-bit, aborting early if there is a mismatch. Only if they match it is necessary to perform two full trial encryptions (the second needs to be done only if the first was successful). It is possible to do more than two full trial encryptions; the running time will hardly be affected. We therefore have:

$$V = 2 \cdot R \cdot \sum_{i=0}^{15} 2^{-i} + 2^{-16}(528 \cdot R + 2^{-32} \cdot 528 \cdot R) < 4.01 \cdot R \quad (2.11)$$

Overall, the cost of the attack is therefore $2^{53.2} \cdot R$, which corresponds to about $2^{44.2}$ full KeeLoq encryptions.

Improvements

Indesteege *et al.* [9, Section 3.4] generalized their attack, making it possible to vary the number of bits of α (t_o) and the number of key bits to determine in step 2 (t_c). The algorithm has to be adapted slightly however. The parameters of the standard algorithm are $t_c = t_o = 16$. The optimal parameters are $t_c = 15$, $t_o = 14$ and offer an improvement of 33% in theory.

Combining the above observation with chosen plaintexts, they could halve the time required to extract the key in practice.

2.3 Side-channel attacks

Side channel attacks (SCA) exploit the weaknesses of the physical implementation of the cryptographic algorithms instead of just the theoretical weaknesses. In this section, two SCA on KeeLoq are briefly presented.

2.3.1 Differential power analysis of the encoding integrated circuits

Eisenbarth *et al.* [7] performed a differential power analysis of Microchip's HCSxxx family of encrypting integrated circuits. They were able to extract the device key k_{dev} by measuring the power consumption of the IC using a resistor and an oscilloscope, and alternatively by measuring the electromagnetic (EM) radiation with a near-field probe.

They were able to extract the key with the resistor with 6 to 30 power traces depending on the packaging of the IC. Using the non-invasive EM measurements, they required 10 power traces in the best case. Their attack considered only the amplitude of the peaks of the measured signal.

Their results are applicable for all key derivation methods and the two protocols (hopping codes and IFF). In case the simple XOR-based (Figure 1.3(a)) key derivation scheme is used, an attacker can extract some/all bits of k_{man} from the device key.

2.3.2 Simple power analysis of software implementation in a micro-controller

Contrary to the encrypting modules, which are implemented in a special-purpose IC, the modules responsible for decrypting are more complex and are usually implemented in a general purpose 8-bit micro-controller (PIC) to perform the decryption. Since these modules have to be able to learn new encrypting modules, they store the manufacturer key k_{man} in read-tamper-proof memory.

In case the more complicated decryption-based key derivation function is used (Figure 1.3(b)), the module can be forced to perform a KeeLoq

decryption with the manufacturer key by entering the "learn" mode and sending random serial numbers.

Using differential power analysis proved to be very time consuming, requiring about 10'000 power traces [7]. In 2009, Kasper *et al.* [10] performed a simple power analysis, which exploited the different number of clock cycles in the computation of the non-linear function on the micro-controller (using the source code given by Microchip). They were able to extract the manufacturer key on a commercial PIC micro-controller.

The attack works even if the decryption process is disturbed by interrupts, as long as it is possible to clearly identify them in the power trace.

Chapter 3

Attacks against the protocol

In this chapter, various attacks against the authentication protocols (hopping codes and IFF), attacks exploiting the key derivation functions, and attacks using cloned keys are described.

3.1 Attacks against the hopping code protocol

In this section some attacks against the hopping code authentication protocol are described.

3.1.1 Exploiting the re-synchronization protocol

Since the range of the counter is only 2^{16}, it is possible to make a workable clone by just gathering 2^{16} plaintexts [9, Section 5.1]. This would require about an hour of effort [2, Section 4.3].

In fact, by taking advantage of the resynchronization protocol, the attacker need only half that number of plaintexts. She would need an oscilloscope and a simple program to record the traces, and probably some sort of mechanism to press the button on the remote for her (a simple Lego construction with a motor). The attacker has to press the button on the remote $2^{15}+2$ times; assuming 10 button presses per second, the whole attack would take 55 minutes. Thanks to the re-synchronization protocol with its rather large re-synchronization window, she only needs to perform a couple of measurement sessions in which she needs to capture two consecutive hopping codes. The goal of the attacker is to store several sets of consecutive hopping codes, such that for any counter value $c \in \mathbb{GF}(2^{16})$, she knows two consecutive hopping codes h_ι and $h_{\iota+1}$ with $c+1 \leq \iota \leq c+2^{15} \pmod{2^{16}}$.

In practice, assuming one key press per second, she would have to perform two measurement sessions at the beginning (h_{init}, h_{init+1}), two sometime in the middle[1] (e.g. $h_{init+\epsilon}$, $h_{init+1+\epsilon}$ with $2 \leq \epsilon \leq 2^{15}$) and two at the

[1]If the attacker has precise control over the number of presses she makes, she would need

end (h_{init+k}, $h_{init+k+1}$ with $2^{15} \leq k \leq \epsilon + 2^{15}$). If she wants to fool the verifier, she now only has to retransmit those six captured messages (cleartext + hopping codes) verbatim, as two consecutive hopping codes will necessarily be within the re-synchronization window, triggering the re-synchronization mechanism.

This attack can very easily be adapted to include more function bits (the attacker has to do captures for all function bits separately). If the manufacturer chose to extend the counter range to 2^{18} bits, this attack can still be done, but will take 7 times longer.

3.1.2 On-line guessing attack

Since the hopping code is 32 bits long, but the open window has a size of 16, and since the counter is not transmitted in the cleartext part of the message, the attacker who purely guesses the hopping code has a success probability of $2^{-32} \cdot 16 = 2^{-28}$ (assuming the attacker has one trace of a legitimate key, so she can spoof the cleartext part of the message). This attack is theoretically faster than any of the off-line attacks, but has not much practical value (at ten guesses per second[2], this attack would take about a year).

3.1.3 Jamming and replay attacks

If the car key transmits out-of-range of the car, the synchronization counter at the decoder will not increment. An attacker can exploit this in two ways:

- By jamming the communication channel when the victim tries to lock his car. The car will then remain unlocked when the victim walks away. This attack, however, has a high probability of being detected by the victim, as cars usually flash their lights and emit a sound when locked.

- By intercepting a signal transmitted when the remote is out of range and the victim is not aware of this (for instance, when the key is in the victim's pocket and is being pressed inadvertently, or if the key is left unattended for several seconds and the attacker has the opportunity to press the buttons), and then replay the captured signal to open the car. However, the captured signal becomes obsolete once the owner uses his key legitimately within range (in effect, re-synchronizing the key and the decoder).

One could also imagine a kind of jamming + replay attack, where the attacker records the legitimate signal, while also jamming the communication channel. This attack is not very practical, since the attacker can typically

only two measurement sessions and measure only $h_{init}, h_{init+1}, h_{init+2^{15}}, h_{init+2^{15}+1}$.

[2]We determined experimentally that the time required to send one message according to the KeeLoq protocol is about 100ms.

only record the signal locking the car (if she want to burgle the car when the victim is leaving his car parked), and she cannot change the function bits (they are part of the ciphertext), making this attack equivalent to the first one.

3.1.4 Partially-known-plaintext attack

None of the cipher-only key recovery attacks presented in the last chapter work in the case of hopping codes, since the attacker does not know the initial value of the synchronization counter. This is no longer a *known plaintext* attack, but only a *partially known plaintext* attack. If she tries to guess the initial counter, the attacks become impractically slow (doing the slide-meet-in-the-middle attack 2^{16} times will require a run time of 2^{61}, where one might be better off doing brute force on FPGA). Under certain circumstances it is conceivable that the attacker might be able to guess a plausible counter range to speed up the search, but the fact remains that there is no good general-purpose method to break the hopping codes.

3.1.5 Extracting the device and master key

The device key can be extracted from the key using differential power analysis and about 10 to 30 power traces. This can be performed by an attacker who has access to the key for a few minutes.

Since the manufacturer key is not saved inside the car key, the attacker can only deduce the manufacturer key if the weak XOR-based key derivation function was used (he can determine the random bits by putting the car key into "learn mode").

If the attacker buys a car, she can try to extract the manufacturer key from the decoder chip. If the chip is a PIC-micro-controller, she can just use simple power analysis; but if the chip is an HCSxxx integrated circuit, she will need prohibitively many power traces (about 10'000).

3.2 Attacks against the IFF protocol

In this section attacks against the IFF authentication protocol are described.

3.2.1 Known-plaintext attack

With about one and a half hours of unsupervised access to the key, the attacker is able to ask the car key to generate the 2^{16} plaintext-ciphertext pairs she needs to crack the device key k_{dev} using the slide-meet-in-the-middle attack. She can then clone the key, and perform some of the attacks described in the next section.

In case the XOR-based key derivation function was used, she can also extract the manufacturer key by putting the car key into "learning mode" to record the seed that was used in the key generation.

3.2.2 Jamming attack

Jamming the communication signal while the user tries to lock his car and walk away also works for the IFF protocol.

3.2.3 Relay attack

If a button on the car key is pressed (inadvertently, or by the attacker in a few seconds of unattended access to the key) and the attacker manages to relay the messages between the car and the key, she can open the car of his victim from an arbitrary distance.

3.3 Attacks with the manufacturer key

Once the attacker knows the manufacturer key corresponding to a given product, it is very easy for her to carry out several damaging attacks with little skill on all cars of the same model. As it was shown in the previous sections, determining the master key is a challenging and/or costly task. It is also very rewarding for a criminal. It is therefore plausible to see this task outsourced to knowledgeable criminal cryptanalysts, who can then sell these manufacturer keys on the black market.

3.3.1 Product piracy

The major reason for deriving the device key from a manufacturer key in the first place is an economic one: customers are forced to buy spare car keys from the original manufacturer at premium prices. For example, the cheapest spare remote for my garage door costs 105 CHF[3] but contains only very simple electronics and a $0.96 KeeLoq HCS300 chip. If the manufacturer key became known, competitors could sell replacement blanks for much cheaper, therefore denying the original manufacturer a very lucrative market.

3.3.2 Extract the device key

The serial number of the key is transmitted with every message, so once the manufacturer key k_{man} is known, only the 0, 32, 48 or 60 bits of randomness introduced during key derivation hinder the cryptanalyst from computing the key k_{dev}. Once k_{dev} is known, the car key can be cloned.

[3]Personal experience with *Novoportes* for a Mini-Novotron 502, 433 MHz remote (excl. costs for installation).

In case no randomness was introduced into the key derivation function, the attacker can deduce the device key with just one eavesdropped communication (to learn the serial number) between the key and the car.

In all other cases, it might be possible to trick the key into entering the "learn mode" and forcing it to reveal the seed it used for key derivation. For this attack to work, the attacker will need to have access to the key for a few minutes. This attack will fail if the manufacturer decided to inhibit the seed transmission.

If 32 bits of randomness were used, the attacker can also launch a remote brute force attack in reasonable time (several seconds with a laptop). Since in IFF, the car key and the car exchange plaintexts and the corresponding ciphertexts, eavesdropping on only one communication should be sufficient (if there are several candidates for a device key, the attacker can just try them out directly with the car). When hopping codes are used, it is more complicated. Only the four function bits are known for sure. The 12 discrimination bits can usually be deduced by looking at the serial number, and the attacker may deduce a relationship between the counter bits of consecutive communications (she never knows them exactly). Two communications fairly close in time are thus necessary for the brute-force attack to have a terminating condition.

If 48 bits of randomness are used, the same method as for 32 bits can be used (the attacker might need two or three eavesdropped communications), but the brute force attack will be much slower (several hours on an FPGA), which is probably not cost effective.

For 60 bits of randomness, the remote attack is not worthwhile.

3.4 Attacks with cloned keys

Finally, attacks that can be performed by an adversary who is in possession of a particular device key are described.

3.4.1 Burgle a car without any signs of break-in

With a cloned remote control, the attacker can open the car of his victim without any sign of break-in, steal the contents of the car, and (optionally) lock the car again. The victim might not notice the missing items since he will find the car locked upon his return, giving the attacker much more time to disappear.

The victim will also have a hard time claiming the theft at his insurance company. The latter will believe the victim was careless and forgot to lock his car.

3.4.2 Steal a car

While Indesteege *et al.* joke that "soon, cryptographers will all drive expensive cars" in the presentation promoting their paper, it must be noted that the door lock and the car immobilizer might be based on two completely different keys. The attacker might have a harder time eavesdropping on the communication when the user turns the ignition (as the range is much shorter), and she might not be able to exploit this effectively (since the user will usually drive away immediately afterwards).

Assuming however that the attacker is indeed able to determine the device key used for disarming the immobilizer, she will be able to drive away with the car.

3.4.3 Denial-of-service attack

The attacker can furthermore perform a kind of denial-of-service attack on garage doors (and to a lesser extent cars[4]) using hopping codes. By transmitting two messages with consecutive counter values at the far end of the "re-synchronization window", she will trigger a re-synchronization at the receiver. The synchronization counter of the legitimate encoder will now be firmly within the "blocked window", thus preventing the legitimate owner of using his remote key anymore (the owner needs to press the button on his key 2^{15} times for it to work again), making him think his remote is broken.

Possible motivations for this include annoying the victim, or even damage the reputation of the manufacturer.

[4]Car keys usually also have a key blade to open the car if the remote fails.

Chapter 4

KeeLoq in real life

According to [2] and [5] (citing [19]), KeeLoq is or has been used by several car manufacturers – including Chrysler, Daewoo, Fiat, GM, Honda, Jaguar, Toyota, Volvo, and Volkswagen – and is used in some garage door openers. In order to verify the claims of the widespread usage of KeeLoq (especially in IFF mode, according to [9]), we have decided to capture and try to reverse-engineer the signal emitted by several car keys (Toyota, Audi, Mercedes-Benz, Volkswagen) and garage door opener (Novoferm[1]).

4.1 Decoding of radio traces

We used an oscilloscope connected to a simple loop antenna to capture the signals. We caught between 1 and 8 million samples per measurement, with sampling frequencies between 50 MS/s and 1.25 MS/s. The anti-aliasing filter of the oscilloscope was disabled because the carrier frequency of the signals was well into the Mega-hertz range.

We then wrote a series of scripts and programs[2] to demodulate and segment the signals (to get the bits on the physical layer), and then extract the data (physical to link layer conversion).

4.1.1 Demodulation

Upon visual inspection with *gnuplot*, the graph of two of the car models (Audi, Mercedes) seemed to be just a single sinusoid of constant frequency and amplitude throughout. After performing a Fast Fourier Transform (FFT) with *GNU Octave*, it is possible to discern two peaks – corresponding to two different frequencies – very close together. The encoding used was therefore a Binary Frequency Shift Keying (BFSK). It was possible to

[1] Mini-Novotron 502, 433 MHz.
[2] In the `radio` directory and its subdirectories.

distinguish the two frequencies in the traces by convolving them with a perfect bandpass filters (the parameters of the filter were determined by visual inspection of the FFT graph of one trace; see Figures 4.2 and 4.3).

The garage door and the other car models (Toyota, VW) used a simple binary amplitude modulation scheme, as we can immediately ascertain by visual inspection of the graphs (see Figures 4.1, 4.4 and 4.5).

4.1.2 Segmentation

Upon visual inspection of the graphs, it was possible to determine the basic bit duration t_e (time element) for each of the car and garage models. It turned out to be 400 μs for Audi and the garage door, and 500 μs for the others. The signal-to-noise ratio for most traces was excellent, so it was enough to write a simple program detecting the transitions between regions in the trace with a high amplitude and a low amplitude: it counts the number of samples with an amplitude higher than a third of the maximum amplitude in a window of size $\frac{t_e}{10}$; if the count drops below some value or becomes larger than some other value[3], a transition is recorded.

For each transition, the program determines the time elapsed between the previous one and rounds it to the nearest integer multiple of t_e. It then outputs the corresponding number of `1`s (if the amplitude was high) or `0`s, forming the physical-layer-bits (see Figure 4.6).

4.1.3 Extracting the data bits

The physical-layer bits all contain a lot of extra information to help the decoder. The transmission starts with a preamble, which is designed "wake-up" the decoder and help synchronize its clock with the encoder. It consists of a series of alternating bits `101010`$\cdots$`10101`, sometimes ended with $\cdots$`010110`. The preamble is usually followed by a header (in the case of KeeLoq, the bit `0` is repeated for $10t_e$ [13]). The data bits are also coded in order to avoid clock drift associated with long sequences of the same bit.

All five models of keys have a preamble of duration between $23t_e$ (garage) and $276t_e$ (Audi). Only the garage key has a header according to the specifications of KeeLoq [13].

There exists several methods to encode the data: Manchester coding (IEEE 802.3[4], G.E. Thomas[5], or differential[6]), Pulse Width Modulation (PWM)[7], to name but the few we came across. Only the garage key used

[3] Hysteresis was necessary to avoid aberrant behavior.

[4] `01` on the physical layer corresponds to `0` on the link layer, while `10` corresponds to `1`.

[5] `10` on the physical layer corresponds to `0` on the link layer, while `01` corresponds to `1`.

[6] Two bits on the physical layer correspond to one bit on the link layer. The second bit of the physical layer is always different from the first. If the first bit of the physical layer is different from the bit that came just before, output `0`, if it is the same output `1`.

[7] `100` on the physical layer corresponds to `0` on the link layer, and `110` corresponds to `1`.

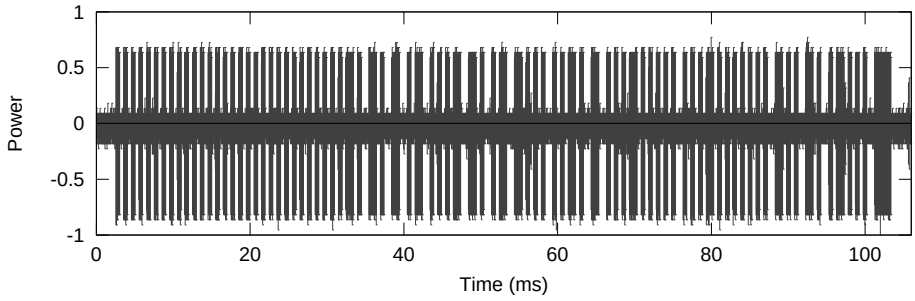


Figure 4.1: A trace from the Toyota key

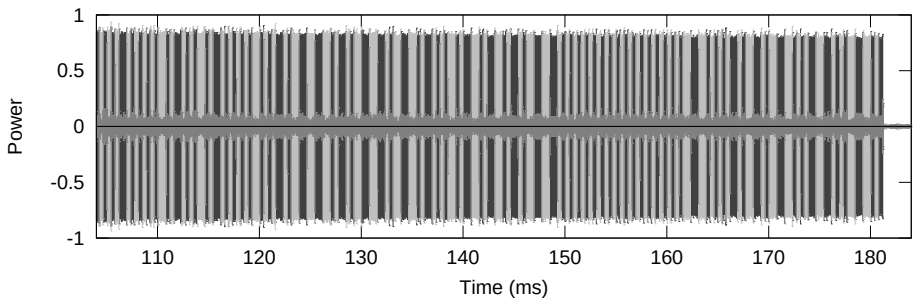


Figure 4.2: Amplitude of an Audi trace after demodulation. The dark and light colors corresponds to the two different frequencies. Parts of the preamble were cut.

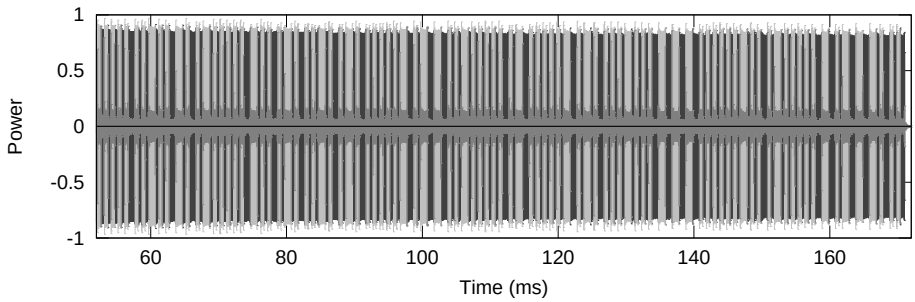


Figure 4.3: Amplitude of a Mercedes trace after demodulation. The dark and light colors corresponds to the two different frequencies. Parts of the preamble were cut.

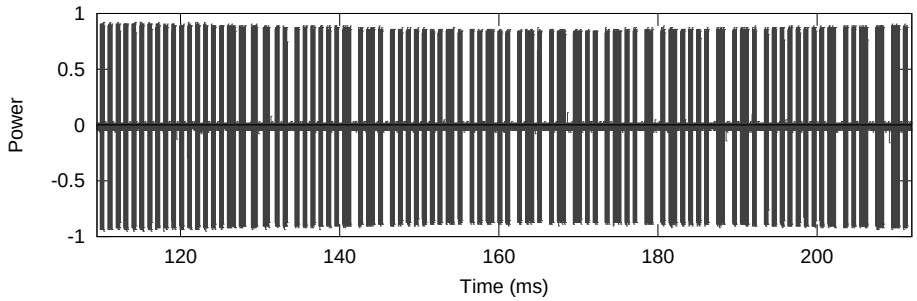


Figure 4.4: A trace from the VW key. Parts of the preamble were cut.

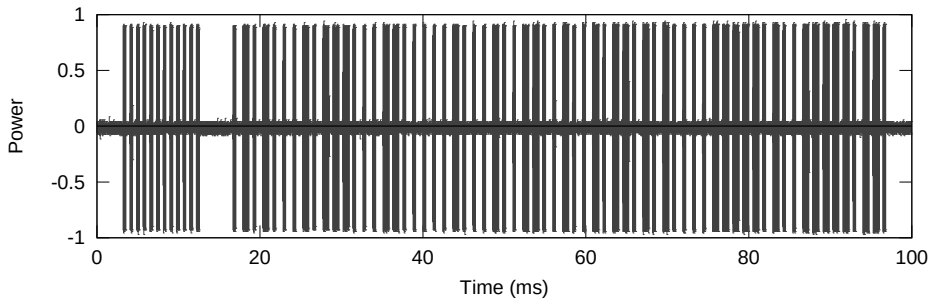


Figure 4.5: A trace from the garage key

```
10101010101010101010101000000000010010011011011011010010010
01101001001001001101001101101101001101101001101101101001001
00110100100100110100100110100110110100110100110100100100100
11011011011011010011010010011011011011011001101101
```

Figure 4.6: The trace shown in Figure 4.5 after segmentation

Binary:
`00111100010000101110110111000100 01001011010100001111101001111101 10`
Hexadecimal: `3C42EDC4 4B50FA7 D 8`

Figure 4.7: The trace shown in Figure 4.6 after PWM decoding. The blocks are: hopping code (32 bits), serial number (28 bits), function (4 bits), battery flag (1 bit) and repeat flag (1 bit).

PWM. All the car keys used non-differential Manchester coding.

All models had a different number of data bits on the link layer: the garage key had only 66, Toyota 69, VW 80, Audi 97, and Mercedes 112. Figure 4.7 shows an example of decoded data.

4.1.4 Interpretation of the link layer

Toyota

The data on the link layer had the following pattern: 4 constant bits over all traces, 65 bits different over all traces.

We suspect that Toyota uses a block cipher of 64 bits, probably also with a hopping code mechanism. All the requisite data is probably inside the plaintext. The last bit might be there for error correction.

Audi

This trace was very confusing to analyze. It did not help that some traces had a very poor signal-to-noise ratio and some traces were not captured from the beginning.

The data on the link layer had the following pattern: 7 fixed bits, 50 random bits[8], 23 constant bits, 17 bits that differed between all traces (there seemed to be a high correlation between certain bits, but it was not perfect). We suspect that Audi uses a block cipher of 48 bits.

Mercedes

This was the longest of all traces.

The data on the link layer had the following pattern: 8 bits that were the same for each button, 5 bits that were different for each trace, 35 constant bits, 64 random bits. The first byte is probably for the function (lock, unlock, open trunk), the second byte seemed to be some sort of counter (since the values were nearly sequential), the next 4 bytes are probably the serial number of the key, and the last 8 bytes probably correspond to the ciphertext of a block cipher (64-bit block size).

VW

The data on the link layer had the following pattern: 8 constant bits, 64 bits that were different for each trace, 1 constant bit, 6 bits that were the same for each button and one constant bit.

Volkswagen probably uses a block cipher of 64 bits. The data probably corresponds to: one byte of serial number, eight bytes of ciphertext, one byte of function code.

[8] Although the first and the last bit were equal for all but one trace.

Garage key

It must be noted that the garage key transmitted two data packets for each key press that differed only in their last bit. A different model of key from the same brand and for the same garage sent identical data packets continuously while the button was held down.

The data on the link layer had the following pattern: 32 bits different over all traces, 28 bits that were constant over all traces, 4 bits that were the same for each button, 1 bit that was always `1`, and the last bit that was `1` in the first transmission and `0` in the second transmission.

This pattern fits exactly with the KeeLoq specifications [13]. The data bits correspond exactly to Figure 1.5. The first flag bit corresponds to the battery low indicator, the second to indicate a repeated transmission.

4.1.5 Conclusion

Every key we tested followed a different standard for the encoding of the physical and link layers.

We have experimentally determined that the two car makers which putatively use KeeLoq in their products seem to have switched to another cipher with longer block size. All car manufacturers we tested except Audi seem to be using a cipher with a 64-bit block size.

The only instance of KeeLoq we encountered in practice was for garage door access control, a much less interesting target. It must be noted that the physical security of the doors themselves leave much to be desired[9], so for the end-user a compromise of KeeLoq is very much of academic interest only.

[9] One can open the main parking door by waving a long stick through the door under the sensor, the decoder of the main door can be reprogrammed from outside, and one can open the individual garages manually with a pair of pliers (or even without tools if the user did not bother to lock the garage with a regular key).

Chapter 5

Software implementation of the attacks

To test the validity of the slide-meet-in-the-middle attack by Indesteege *et al.* [9] and the slide-algebraic attack by Courtois *et al.* [5, Section 7.2], we decided to implement the attacks. You can find the source code and demonstration in the `prog` directory.

You can type `make` to compile everything in this directory. However, if your processor is not a Core2, you must adapt (or remove) the string `-mtune=core2` from the `CXXFLAGS` in the `Makefile`.

The programs were written and tested on a Dell Inspiron 6400 laptop with an Intel Core2 T7200 CPU (2GHz, 4MiB L2-cache, only 1 of 2 cores used), 2 GiB of main memory (2 GiB of swap). The operating system used was Ubuntu Karmic Koala. Run times were measured either with the Unix *time(1)* program, or directly from the program output if it displays the total run time (MiniSat and CryptoMiniSat).

5.1 Slide-meet-in-the-middle attack

We implemented the attack by Indesteege *et al.* according to both the standard algorithm [9, Section 3.3] and the generalization with optimal parameters [9, Section 3.4] (*cf.* Section 2.2.5).

5.1.1 Implementation

Source code: `preneel.cpp` and `preneel2.cpp`.
Demonstration: can be found in the `test_preneel` subdirectory. The scripts used for the measurements were `preneel.sh` and `preneel2.sh`.

There were no major difficulties encountered in the implementation, as the paper provided good explanations and included pseudocode that could be quickly translated to a working program.

Notes

It is very strongly recommended to run this program on a computer with at least 3 MiB resp. 4 MiB of L2-cache, since the performance of the program critically depends on the speed of retrieval of elements in a hash table.

Our program expects a list of 2^{16} plaintext-ciphertext pairs. The program `genkp.cpp` will generate such a list, and can also check whether the list includes a slid pair.

In order to speed up computation, the first 16 bits of the key may be provided. For even faster computation, a guess for α might also be provided (`genkp.cpp` can print out α corresponding to a particular slid pair); however, the comparison between the two variants will no longer be fair.

5.1.2 Results

The measured run time of both approaches is summarized in the Table 5.1:

Variant	Run time	Time for full attack
Standard ($t_c = t_o = 16$)	51'38"304	2350 days
Optimal ($t_c = 15$, $t_o = 14$)	50'18"768	2290 days

Table 5.1: Measured worst-case run time for both variants (the program does not halt when it finds the key), when the first 16 bits of the key are provided. Since these are worst-case measurements, the run time is almost constant over different trials; so we decided to make only one trial. The time for the full attack is computed by multiplying the run time by 2^{16}.

The optimal variant is just 2.8% faster than the standard approach, and not 33% as expected in theory. We surmise that this is because the optimal program needs to do much more bit manipulations than the standard approach.

The program would take $\frac{2290 \text{ days}}{\text{available 2GHz CPU cores}}$ (worst case) if no key bits were provided (for 82 quad-core CPUs this would take one week). Since our computer needs about 4.7 μs to do one full KeeLoq encryption, the worst-case run time of our program is $2^{45.26}$ KeeLoq encryptions.

When corrected for differences in CPU speed, our program runs just 2.25 times slower that the implementation of Indesteege *et al.* [9, Section 4].

5.2 Slide-algebraic attack

We also implemented the slide-algebraic attack [5, Section 7.2] by Courtois *et al.*

5.2.1 SAT solvers and Boolean normal forms

The slide and algebraic attack requires the use of a SAT solver, which can be considered as a "black box". SAT solvers solve the *Boolean satisfiability problem* (SAT): given a Boolean formula in Conjunctive Normal Form (CNF), find an assignment of variables such that the whole formula evaluates to TRUE if one exists, else output "unsatisfiable". SAT is a well-known NP-complete problem, so any algorithm that solves it must have an exponential running time[1]. However, practical implementations are often able to solve systems with up to thousands of variables and clauses in seconds.

Conjunctive Normal Form

A formula is in CNF if it consists of a conjunction of clauses (i.e., clauses separated by logical-AND operators). A clause consists of a disjunction of literals (literals separated by logical-OR operators). A literal is either a variable or a negated variable.

A Boolean formula can always be transformed to CNF by applying distributivity and De Morgan's laws. For example, Equation 5.7 is in CNF.

Some SAT solvers, such as CryptoMiniSat, also accept clauses which consist of literals separated by XOR-operators. This is convenient, since it frees the programmer from converting an XOR-clause consisting of n literals into 2^{n-1} regular clauses.

Algebraic Normal Form

Another widely used Boolean normal form is the Algebraic Normal Form (ANF). An equation is in ANF, if the right hand side is a sum of monomials (and each monomial is a product of variables), i.e., if it can be written as:

$$\begin{aligned} f(x_1, x_2, \cdots, x_n) = a_0 + \\ & a_1 x_1 + a_2 x_2 + \cdots a_n x_n + \\ & a_{1,2} x_1 x_2 + a_{1,3} x_1 x_3 + \cdots + a_{n-1,n} x_{n-1} x_n + \\ & \cdots + \\ & a_{1,2,\cdots,n} x_1 x_2 \cdots x_n \end{aligned} \tag{5.1}$$

For instance, Equation 1.9 is in ANF.

5.2.2 A mistake in Courtois *et al.*'s paper

According to Courtois *et al.* [5, Section 7.2], if you are given a slid pair ($P^{[j]} = f_k(P^{[i]})$), it is sufficient to write the equations for 64 rounds for the

[1] Assuming $P \neq NP$.

two slid pairs:

$$\begin{aligned} P^{[j]} &= f_k(P^{[i]}) && \text{(64 equations)} \\ C^{[j]} &= f_{k_{15..0,63..16}}(C^{[i]}) && \text{(64 equations)} \end{aligned} \tag{5.2}$$

In ANF there are 128 equations, 64 unknowns from the key and 2×32 unknowns from the intermediate values of the encryption state register. However, since the system is non-linear, it is possible that there are several solutions. When running the attack, CryptoMiniSat outputs a wrong key about half the time. For example for the following parameters:

$$\begin{aligned} k &= \texttt{0x5cec6701b79fd949} \\ P^{[i]} &= \texttt{0xf741e2db} \end{aligned}$$

We have:

$$\begin{aligned} P^{[j]} &= \texttt{0xca69b92} \\ C^{[i]} &= \texttt{0xe44f4cdf} \\ C^{[j]} &= \texttt{0xa6ac0ea2} \end{aligned}$$

The SAT solver may however output:

$$k' = \texttt{0x5cef6603971dd949} \neq k$$

Indeed we have:

$$\begin{aligned} P^{[j]} &= f_k(P^{[i]}) = f_{k'}(P^{[i]}) \\ C^{[j]} &= f_{k_{15..0,63..16}}(C^{[i]}) = f_{k'_{15..0,63..16}}(C^{[i]}) \end{aligned} \tag{5.3}$$

The mistake lies in the fact that:

$$E_k(P^{[i]}) = C^{[i]} \neq \texttt{0xbbb828bc} = E_{k'}(P^{[i]}) \tag{5.4}$$

which was not explicitly stated in the input.
To avoid this problem, we need to write the 464 missing equations bridging $P^{[j]}$ to $C^{[i]}$. The system then becomes:

$$\begin{aligned} P^{[j]} &= f_k(P^{[i]}) && \text{(64 equations)} \\ C^{[i]} &= E_k(P^{[i]}) && \text{(528 equations, but 64 are the same as above)} \\ C^{[j]} &= f_{k_{15..0,63..16}}(C^{[i]}) && \text{(64 equations)} \end{aligned} \tag{5.5}$$

for a total of $592 = 64 + 528$ equations (in ANF).

5.2.3 Implementation

Source code: `keeloq_cnf.cpp` (outputs the equations in CNF, the output must be fed to CryptoMiniSat).
Demonstration: can be found in the `test_courtois` subdirectory. The scripts `test_courtois.sh` and `test_courtois_negative.sh` will do everything for you. You may pass a list of options to the scripts to test out the different variants (run `../keeloq_cnf -h` for a list of options).
Dependencies: This requires the program CryptoMiniSat version 1.2.11 [16]. The source code and binaries are included in the deliverables. Alternatively, it is possible to use the program MiniSat2 [6].

Unlike the previous attack, no explicit algorithm was given in the paper, so we had to devise the equations to be input to the SAT-solver ourselves. We initially used CryptoMiniSat as a SAT-solver, since it allows XOR clauses. Later, we also used MiniSat2, the SAT-solver that was used by Courtois *et al.* [5, Section 6.4].

The equations given in Section 1.1.1 are in ANF, and had to be converted to CNF-with-XOR before they could be solved by CryptoMiniSat. We did not find a satisfactory ANF-to-CNF converter, so we decided to write one ourselves. We proceeded in three steps:

- **Step 1:** Write the equations as they appear in Equations 1.9 and 1.12 (using CNF-with-XOR clauses). The only nontrivial step was to handle the multivariate monomials inside the NLF:

  For instance, we can replace the monomial ade by an auxiliary variable γ, and write the additional clauses corresponding to $\gamma = ade$:

  $$(\overline{\gamma} \vee a) \wedge (\overline{\gamma} \vee d) \wedge (\overline{\gamma} \vee e) \wedge (\gamma \vee \overline{a} \vee \overline{d} \vee \overline{e}) \tag{5.6}$$

- **Step 2:** Keep the encryption state equations (Equation 1.12) in CNF-with-XOR, but transform the equations of the NLF in CNF (with a 6 variable Karnaugh map). Equation 5.7 below shows the equivalence of $f = \mathrm{NLF}(a, b, c, d, e)$ in CNF:

  $$\begin{aligned}
  &(\overline{f} \vee \overline{c} \vee e \vee a \vee b) \wedge (\overline{f} \vee c \vee \overline{e} \vee a \vee \overline{b}) \wedge (f \vee \overline{c} \vee d \vee \overline{b}) \\
  \wedge\ &(\overline{f} \vee \overline{d} \vee e \vee \overline{a} \vee \overline{b}) \wedge (f \vee \overline{c} \vee e \vee \overline{a} \vee b) \wedge (\overline{f} \vee \overline{c} \vee \overline{d} \vee \overline{e}) \\
  \wedge\ &(f \vee a \vee b \vee d \vee \overline{e}) \wedge (f \vee a \vee \overline{b} \vee \overline{d} \vee e) \wedge (\overline{f} \vee c \vee d \vee e) \\
  \wedge\ &(f \vee d \vee \overline{e} \vee \overline{a} \vee \overline{b}) \wedge (f \vee c \vee \overline{d} \vee \overline{e} \vee \overline{a}) \wedge (f \vee c \vee \overline{d} \vee b) \\
  \wedge\ &(\overline{f} \vee d \vee \overline{e} \vee \overline{a} \vee b)
  \end{aligned} \tag{5.7}$$

- **Step 3:** Write everything in regular CNF by breaking up the 5-term XOR of Equation 1.12 into 16 clauses. After this step, we can also run MiniSat.

A comparison of the number of clauses and variables generated can be found in Table 5.2.

		Courtois' version		Corrected version	
Step	XOR clauses	Variables	Clauses	Variables	Clauses
1	yes	1 536	4 608	6 608	20 848
2	yes	384	1 920	**1 280**	**8 416**
3	no	384	3 840	1 280	17 296

Table 5.2: Number of clauses and variables in the problem submitted to the SAT solver for the slide-algebraic attack after the different steps

5.2.4 Experimental results

We simulated 100 runs of the attack for each setting, since the run time of SAT solvers is highly variable and log-normally distributed [1, Section 13.5.3]. The following settings were tested: running the corrected and uncorrected variants; after each step in the ANF-to-CNF conversion; when given a slid pair (positive run) or random plaintexts (negative runs). Everything was tested with CryptoMiniSat 1.2.11, and step 3 was also tested with MiniSat 2.0 (beta). The run times are summarized in Table 5.3.

		Positive test			Negative test		
Step	Program	μ_{time}	σ_{time}	OK	μ_{time}	σ_{time}	OK
Courtois *et al.*'s version: (128 equations)							
1	cryptominisat	7.12 s	6.74 s	53%	17.2 s	9.62 s	50%
2	cryptominisat	1.30 s	1.12 s	57%	2.19 s	1.56 s	43%
3	cryptominisat	1.34 s	1.16 s	57%	2.27 s	1.26 s	54%
3	minisat2	1.16 s	1.10 s	51%	2.25 s	1.41 s	47%
Corrected version: (560 equations)							
1	cryptominisat	9.50 s	6.52 s	100%	20.7 s	7.43 s	100%
2	cryptominisat	**1.87 s**	1.25 s	100%	**3.63 s**	1.46 s	100%
3	cryptominisat	2.12 s	1.39 s	100%	3.92 s	1.81 s	100%
3	minisat2	2.23 s	1.76 s	100%	4.40 s	2.09 s	100%

Table 5.3: Run time of the slide-algebraic attack, with a variety of parameters. μ_{time} is the average run time over 100 runs, σ_{time} is the standard deviation, OK is the probability the algorithm found the original key.

Assuming that all 2^{32} instances can be run in parallel and that the program halts as soon as it finds a solution (in practice, due to caching, all instances should be run for some time, and if they did not complete they should be put to sleep and the next unfinished instance should be run [4]), we do not need to care about the run time of unsatisfiable instances. The total run time would be $\frac{254.51 \text{ years}}{\text{available 2GHz CPU cores}}$. On our computer, performing a full KeeLoq encryption takes 4.7 μs, so the run time of the full attack is equivalent to $2^{50.6}$ KeeLoq encryptions (which is a bit faster than Courtois *et al.*'s results [5, Section 7.2]).

Chapter 6

Improving the attacks against KeeLoq

In this chapter, several attempts to improve the attacks against KeeLoq are described. We have first tried to find a better bias in the NLF; next, we tried to find ways to adapt the attacks against KeeLoq to work for the hopping code protocols; and finally we experimented with Gröbner bases, an alternative to SAT solvers.

6.1 Finding a better bias in the NLF

The slide-correlation attack relies on the fact that the NLF has a bias of $\frac{1}{8}$ [2]:

$$\Pr[\mathrm{NLF}(a, b, c, d, e) = d \oplus e] = \frac{1}{2} + \frac{1}{8} \tag{6.1}$$

We have found a better bias (see `findbias.cpp`):

$$\Pr[\mathrm{NLF}(a, b, c, d, e) = c \oplus d\,] = \frac{1}{2} + \frac{1}{4} \tag{6.2}$$

Regrettably, this bias cannot be used to improve Bogdanov's attack, since the latter relied on the fact that the NLF could be approximated by the first 16 bits of the text register (d and e represent positions 9 and 1 of the text register, while c represents bit 20).

The better bias does not help with linear or differential cryptanalysis either, because of the prohibitively high number of rounds of the cipher.

6.2 Adapting the slide-algebraic attack to hopping codes

In Section 3.1.4 it was observed that all of the published key recovery attacks work for the IFF protocol. It was also observed that simply guessing the

initial counter value and then running the slide-meet-in-the-middle attack, one could break the hopping code protocol in 2^{61} KeeLoq encryptions. In this section, we detail how we tried to break the hopping code protocol by adapting the slide-algebraic attack.

6.2.1 Model

The attacker gathers 2^{16} consecutive hopping codes with the same 16 MSB in each plaintext. Since the 16 MSB contain only public information (serial number and button status) which are either easily obtainable, or transmitted in cleartext with each message, we assume she knows the 16 MSB.

We assume that the attacker does not know the initial counter value, so she does not know the 16 LSB of each plaintext. However, since she gathers consecutive codes, she knows the difference between the 16 LSB of each pair of plaintexts.

6.2.2 Implementation

Source code: `keeloq_anf.cpp` (with the `-p` option).
Demonstration: `test_hopping.sh` in the subdirectory `test_courtois`.

We slightly modify the slide-algebraic attack: two plaintexts are taken at random and assumed to be a slid pair. The ciphertext, the 16 MSB of each plaintext, and the 560 equations shown in Equation 5.5 are written. Additionally, the following equation is written:

$$L_{79..64} = L_{15..0} + \delta \pmod{2^{16}} \tag{6.3}$$

Where δ is known and equal to $L_{79..64} - L_{15..0} \pmod{2^{16}}$. The above equation can be easily translated to CNF by using the equations for a 16-bit full adder and with the help of a 4-bit Karnaugh map.

We only ran the attack with slid pairs. As mentioned previously, the probability of having a slid pair in 2^{16} plaintexts is 63%.

6.2.3 Results

We used the same setup as in Chapter 5. We used CNF-with-XOR (Step 2 in Section 5.2.3). Since running the full attack takes much too long, we decided to leak some key bits[1].

The results are summarized in Table 6.1. It is surprising that the more key bits we guess, the faster the full attack becomes. Overall the results are disappointing since running the original slide-algebraic attack and guessing the initial counter value is about 7 times faster (but still slower than brute force).

[1]Of course, for each key bit guessed, the complexity of the full attack doubles.

Key leaked	Trials	μ_{time}[a]	σ_{time}[b]	Med_{time}[c]	Full attack[d]
22 bits	400	0.212 s	0.114 s	0.220 s	$\mathbf{2^{69.461}}$
20 bits	100	0.892 s	0.518 s	0.912 s	$2^{69.534}$
18 bits	25	4.08 s	2.55 s	3.86 s	$2^{69.727}$
16 bits	10	18.3 s	12.1 s	22.5 s	$2^{69.893}$
14 bits	10	99.6 s	60.7 s	123 s	$2^{70.337}$
12 bits	10	513 s	315 s	595 s	$2^{70.702}$
Original[e]	100	1.87 s	1.25 s	1.67 s	$2^{66.602}$

[a] Average run time.

[b] Standard deviation of the run time.

[c] Median run time.

[d] Run time of the full attack, in full KeeLoq encryptions (one encryption takes 4.7 μs).

[e] Using the original slide-algebraic attack and guessing $L_{15..0}$ instead of the key.

Table 6.1: Run time of the extension of the slide-algebraic attack

6.3 Gröbner Bases

Gaussian Elimination can be used to solve a linear system over $\mathbb{GF}(2)$. KeeLoq and many other ciphers, however, can only be described by a system of non-linear equations. As was stated before, SAT solvers can be used to solve these systems numerically, but they typically only output one solution. Another method is to use Buchberger's algorithm or the Faugère F4 algorithm, which both compute the Gröbner basis G of a system of non-linear polynomials I over any field, such as $\mathbb{GF}(2)$ [18]. The Gröbner basis G preserves all roots of I, and so it can be used to quickly compute all solutions. The time required to compute a Gröbner basis is highly variable (see Section 6.3.2), depends on the specific implementation [1, Section 15.7] and may take a lot of memory (usually it crashes due to insufficient memory [1, Section 15.7]). The monomial ordering (i.e. which variables appear first in the solution) chosen also has a huge impact on the run time [18]. Several computer algebra systems, such as Magma, Singular [8], or PolyBoRi [3], implement algorithms for computing Gröbner bases.

For some systems of equations, computing the Gröbner basis is much faster than using SAT solvers [1, Section 15.7]. Courtois *et al.* [5, Section 6.3] experimented with Singular's implementation of Gröbner bases, but with little success. We therefore decided that it was worthwhile to try to break KeeLoq with Gröbner bases.

6.3.1 Implementation

Source code: `keeloq_anf.cpp` (by default outputs a PoliBoRi script, with the `-S` option it outputs a SINGULAR script).
Demonstration: `groebner_*.sh` in the subdirectory `test_groebner`.

Several tests were performed with both PolyBoRi and SINGULAR[2]:

- **Test 1:** Given a random plaintext and key, compute the ciphertext (this can be though as a sort of "unit test" to make sure that the equations were written correctly).
- **Test 2:** Given a single known plaintext/ciphertext pair (KP), recover the key.
- **Test 3:** Given two known plaintext/ciphertext pairs, recover the key.
- **Test 4:** The original version of the slide-algebraic key-recovery attack (c.f. Section 5.2.2).

The polynomials for which the Gröbner basis was to be computed on, were taken from Equation 1.12, which was already in the correct format. The degree reduction technique mentioned in Courtois *et al.*'s paper [5, Section 6.1] was used to write the NLF. Furthermore, all known variables were directly substituted in the equations.

6.3.2 Results

Like in the case of SAT solvers, the run time to compute Gröbner bases is highly variable. Also, the computations need a huge amount of memory. We therefore limited the run time of all computations to 20 minutes and the memory to 2 GiB. The run time for PoliBoRi is summarized in Table 6.2, and the results for SINGULAR's `slimgb` function[3] in Table 6.3.

SINGULAR tends to allocate a lot of memory, and therefore frequently ran out of memory on the larger problems after one to five minutes. Even if there is a single solution, `slimgb` tends to output a Gröbner basis in *non-reduced* row echelon form, which makes parsing the solution harder. In contrast, PoliBoRi tends to use much less memory, and frequently ran out of time on the larger problems. If the solution is unique, PoliBoRi always outputs the Gröbner basis in reduced row echelon form. PoliBoRi's performance is slightly better than SINGULAR's, but this is not a surprise as PoliBoRi was optimized for computations over Boolean rings (hence the name).

Compared to SAT solvers, the performance of the Gröbner basis computation is appalling. The results of test 2 and 3 strongly indicate that direct

[2] MAGMA was not included due to lack of availability.
[3] We also tried using the `groebner` function, but its performance was much worse than `slimgb`.

Rounds	Trials	Failed[a]	Unique[b]	μ_{time}	σ_{time}	Med_{time}
Test 1: (Encryption)						
528	25	0%	100%	29.2 s	0.489 s	29.1 s
Test 2: (Key recovery with 1 KP)						
64	25	0%	0%	0.315 s	0.006 s	0.31 s
68	25	0%	0%	3.34 s	1.89 s	2.72 s
72	10	50%	0%	–	–	444 s
Test 3: (Key recovery with 2 KP)						
52	25	0%	44%	2.23 s	6.91 s	0.53 s
56	10	30%	43%	–	–	161 s
Test 4: (Key recovery attack with a slid pair)						
48	25	0%	28%	45.1 s	98.8 s	8.56 s
52	6	100%	–	–	–	–

[a]Percentage of trials which didn't complete after 20 minutes.

[b]Percentage of non-failed trials which output a unique solution to the system.

Table 6.2: Testing the Gröbner basis algorithm with PolyBoRi. Tests 2 to 4 are all round-reduced, since running a full attack would take too long.

Rounds	Trials	Failed[a]	μ_{time}	σ_{time}	Med_{time}
Test 1: (Encryption)					
528	10	0%	197 s	6.92 s	196 s
Test 2: (Key recovery with 1 KP)					
44	25	0%	4.94 s	14.5 s	0.25 s
48	3	100%	–	–	–
Test 3: (Key recovery with 2 KP)					
48	25	0%	0.032 s	0.011 s	0.03 s
52	10	20%	–	–	18.8 s
56	4	100%	–	–	–
Test 4: (Key recovery attack with a slid pair)					
44	25	0%	0.162 s	0.312 s	0.06 s
46	10	30%	–	–	15.4 s
48	4	100%	–	–	–

[a]Percentage of trials which ran out of memory.

Table 6.3: Testing the Gröbner basis algorithm with Singular's `slimgb` function. Tests 2 to 4 are all round-reduced, since running a full attack would take too much memory.

key recover for full-round KeeLoq is slower than bruteforce. The results of test 4 show that a direct key recovery attack using Gröbner basis and our method is slower than using SAT-solvers and slower than brute force.

Chapter 7

A new key schedule

In this chapter, we will present a new key schedule for KeeLoq that will render the cipher immune to slide attacks.

7.1 Resistance against slide attacks

The prerequisite for launching a slide attack is that the key schedule repeats:

$$\exists s > 0 \quad \forall i, s \leq i < 528 \quad K_0^{\langle i-s \rangle} = K_0^{\langle i \rangle} \tag{7.1}$$

For the regular KeeLoq key schedule (which is vulnerable to slide attacks), this equations is satisfied for $s = 64$.

7.2 Description of the new key schedule

Let $r_{527..0} \in \{0,1\}^{528}$ be a vector of 528 numbers. This vector shall be part of the description of the cipher.

In each round i, a binary rotation of the key register $r_i + 1$ bits to the right shall be performed:

$$K_{63..0}^{\langle i+1 \rangle} = \mathrm{ROR}_{r_i+1}(K_{63..0}^{\langle i \rangle}) \tag{7.2}$$

The rest of the cipher shall operate as per Equations 1.4, 1.6 and 1.8.

Intuitively, the key register is shifted either one or two bits to the right. This is somewhat similar to the key schedule of the Data Encryption Standard (DES) [17, p. 25].

7.2.1 Choosing the parameter r

The resistance of this new key schedule depends critically on r: indeed, if all elements of r are `0`, the regular KeeLoq key schedule is performed.

Case 1: Full random

One possibility would be to choose each element of r independently at random. There are two problems with this approach:

- First, it unnecessarily complicates the description of the cipher. Hardware implementations will need much more gates, and software implementations will need to store at least an extra 528 bits (66 bytes).
- Second, with high probability, the usage of the key bits will be unevenly distributed (see Table 7.1).

In what follows, $r_{527..0} = \lfloor (\pi - 3) \cdot 2^{528} \rfloor$ will be used as a representative[1] of a random number (case 1).

Case 2: Partial random

A better method, would be to choose only the first R elements of r at random, and then "recycle" the old numbers: $\forall i \geq R \quad r_i = r_{i-R}$. The description of the cipher will be much simpler, as it requires the storage of only R extra bits of information.

We can again use the digits of π as our random number: for case 2 we have chosen to use $R = 31$ bits.

Case 3: Optimize for smallest R

An even better approach would be to quit using random numbers, and instead try to find the smallest R (with corresponding r) such that these essential properties are satisfied:

- All key bits must be used.
- It must not be possible to mount a slide attack on the key schedule.

We have determined experimentally that for $R \leq 8$ there exists no r which satisfies these properties[2], and that for $R = 9$ there are 246 satisfactory values of $r_{8..0}$: we shall use the first one $r = 000000011_2 = \texttt{0x03}$ as representative for case 3.

[1] In this definition, π is the well-known mathematical constant $3.141592\cdots$. The constant r is not really random, but is a "nothing up my sleeve number": a number exhibiting good random-like properties, but which is "above suspicion of hidden properties" (Wikipedia).

[2] After $64 \cdot R$ rounds, the key register must necessarily be the same as in round zero, so it must be possible to launch a slide attack with window size $s = 64 \cdot R$.

Key schedule	R	Min	Max	St. Dev	Slide attacks
Regular KeeLoq	1	8	9	0.433	yes: $s = 64$
Case 1[a]	528	4	12	1.7	no
Case 2[b]	31	7	10	0.935	no
Case 3[c]	9	8	9	0.433	no

[a] $r_{527..0} = \lfloor(\pi - 3) \cdot 2^{528}\rfloor =$ `0x243F6A8885A308D2A82B5E7BAC1594F52A`$\cdots$

[b] $r_{30..0} = \lfloor(\pi - 3) \cdot 2^{31}\rfloor =$ `0x121FB544`

[c] $r_{8..0} =$ `000000011`$_2 =$ `0x03`

Table 7.1: Comparing the distribution of the key bits (number of occurrences of the least and most frequently used key bit, standard deviation on the number of occurrences of each key bit), and susceptibility to slide attacks, for regular KeeLoq and the three variants of the proposed key schedule

7.3 Performance

Source code: `speed.cpp` and `keeloq_keysched.cpp`.

We compared the performance of the 3rd key schedule ($r_{8..0} =$ `0x03`) with the regular one. We used the implementation by Ruptor [15] for the latter. The setup was the same as in chapter 5. The source code was compiled both will full compiler optimizations, and without any compiler optimizations. The results are presented in Table 7.2.

The new key schedule performs about as well as the old one. We can see that with full optimization, the decryption function of the new key schedule is noticeably slower; we surmise that this is due to caching effects of the CPU and has no importance for implementation on more modest hardware.

	`-O3` optimization		`-O0` optimization	
Key schedule	Encryption	Decryption	Encryption	Decryption
Old	4.651 μs	4.527 μs	8.641 μs	8.205 μs
New	4.890 μs	7.360 μs	8.509 μs	8.448 μs

Table 7.2: Comparing the performance of the old and new key schedule, both compiler-optimized (`-O3`) and non-optimized (`-O0`). We ran one million encryptions and decryptions for each scenario.

7.4 Conclusion

For the improved key schedule, case 3, viz. $R = 9$ $r_{8..0} =$ `000000011`$_2$, shall be used. We can see from Table 7.1 that this alternative offers protection against slide attacks, has a key usage distribution similar to that of the regular KeeLoq, and requires the least amount of extra resources in software and hardware implementations.

Conclusion

Although there are claims of widespread usage of KeeLoq in the literature [2, 5, 7, 9, 10, 19], we were able to identify the usage of KeeLoq only in a garage door opener, but not in any of the cars key we analyzed — all manufacturers we analyzed seem to have switched to ciphers with longer block length. Furthermore, all remotes seemed to use single-way communication, casting some doubt over the widespread usage of the IFF protocol.

We found out that implementing the slide-meet-in-the-middle attack — the fastest attack which does not require an unrealistic data gathering effort — is very easy, and that performing the full attack is feasible with modest hardware. We also found that the slide-algebraic attack as published by Courtois *et al.* [5, Section 7.2] was incomplete, and that the published run time is difficult to achieve in practice due to the parallelization requirements.

We have tried to improve the attacks against KeeLoq, especially to make a workable key recovery attack against hopping codes, but we could not find anything faster than the slide-meet-in-the-middle and guessing the initial counter value (2^{61} KeeLoq encryptions).

Finally, we have seen that it is possible to slightly change the key schedule of KeeLoq to make it immune against slide attacks, making it invulnerable to all currently published attacks. This change has very little performance penalty.

Future work

We believe that the potential of slide attacks against KeeLoq in IFF mode has been exhausted. No satisfactory attack against KeeLoq in the prevalent hopping code mode exists yet, so this problem might be of interest for future research.

Our survey of car and garage keys comprised only five brands; in order to determine the full extent of KeeLoq usage, radio traces of more manufacturers are needed.

Theoretical and practical knowledge gained

For me, it was an interesting experience to study a subject in detail: applying the ideas gathered in the litterature analysis, implementing the attacks in software and trying out variations. Seeing a real-world cipher in action and studying a new form of cryptanalysis in detail by myself was a very gratifying. It was also the first time I was confronted with actual radio traces, which I decoded by trying various techniques lernt a few semesters ago. Frustration and disapointment was also involved, especially when my ideas turned out to be worse when tried out. Overall, the most valuable gain was to get a good glimpse of what reasearch in cryptography is.

Bibliography

[1] G.V. Bard. *Algebraic Cryptanalysis*. Springer-Verlag New York Inc, 2009.

[2] A. Bogdanov. Linear slide attacks on the KeeLoq block cipher. *Lecture Notes In Computer Science*, 4990:66–80, 2008.

[3] M. Brickenstein and A. Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, 2009.

[4] N.T. Courtois. Personal communication, December 9, 2009.

[5] N.T. Courtois, G.V. Bard, and D. Wagner. Algebraic and slide attacks on KeeLoq. *Lecture Notes In Computer Science*, 5086:97–115, 2008.

[6] N. Een and N. Sörensson. MiniSat v2. 0 (beta). *Solver descriptions, SAT Race*, 2006, 2006.

[7] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M.T.M. Shalmani. On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme. In *Advances in Cryptology-CRYPTO*, volume 5157, pages 203–220. Springer, 2008.

[8] G.M. Greuel, G. Pfister, and H. Schönemann. Singular 3-1-0. `http://www.singular.uni-kl.de/`, 2009.

[9] S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel. A practical attack on KeeLoq. *Lecture Notes in Computer Science*, 4965:1, 2008.

[10] M. Kasper, T. Kasper, A. Moradi, and C. Paar. Breaking KeeLoq in a Flash: On Extracting Keys at Lightning Speed. *AFRICACRYPT 2009*, 5580:403–420, 2009.

[11] Microchip. *Secure learning RKE systems using KeeLoq encoders (TB001)*, 1996. `http://ww1.microchip.com/downloads/en/AppNotes/91000a.pdf`.

[12] Microchip. *Code hopping decoder using a PIC16C56 (AN642)*, 1998. http://www.keeloq.boom.ru/decryption.pdf.

[13] Microchip. *HCS300 KeeLoq code hopping encoder*, 2001. http://ww1.microchip.com/downloads/en/DeviceDoc/21137f.pdf.

[14] Microchip. *Using KeeLoq to validate subsystem compatibility (AN827)*, 2002. http://ww1.microchip.com/downloads/en/AppNotes/00827a.pdf.

[15] Ruptor. KeeLoq C source code. http://cryptolib.com/ciphers/keeloq/.

[16] M. Soos, K. Nohl, and C. Castelluccia. Cryptominisat v.1.2.11. http://planete.inrialpes.fr/~soos/CryptoMiniSat/index.html, 2009.

[17] S. Vaudenay. *A classical introduction to cryptography: applications for communications security*. Springer-Verlag New York Inc, 2005.

[18] Gröbner Basis Wikipedia article. http://en.wikipedia.org/wiki/Gr%C3%B6bner_basis, December 2009.

[19] KeeLoq Wikipedia article. http://en.wikipedia.org/wiki/KeeLoq, December 2009.