Exposing iClass Key Diversification

Flavio D. Garcia Gerhard de Koning Gans Roel Verdult

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands.
{flaviog,gkoningg,rverdult}@cs.ru.nl

Abstract

iClass is one of the most widely used contactless smartcards on the market. It is used extensively in access control and payment systems all over the world. This paper studies the built-in key diversification algorithm of iClass. We reverse engineered this key diversification algorithm by inspecting the update card key messages sent by an iClass reader to the card. This algorithm uses a combination of single DES and a proprietary key fortification function called 'hash0'. We show that the function hash0 is not one-way nor collision resistant. Moreover, we give the inverse function hash0^{-1} that outputs a modest amount (on average 4) of candidate pre-images. Finally, we show that recovering an iClass master key is not harder than a chosen plaintext attack on single DES. Considering that there is only one master key in all iClass readers, this enables an attacker to clone cards and gain access to potentially any system using iClass.

1 Introduction

Over the last few years, much attention has been paid to the (in)security of the cryptographic mechanisms used in contactless smartcards [NESP08, GdKGM+08, GvRVS09, COQ09, GvRVS10].

This paper does not focus on the security of the cards themselves but on the security of the cryptographic protocols used in the embedding systems. Concretely, we study the key diversification and the proprietary 'key fortification' functions of the HID iClass contactless smartcards and the secure key loading mode of the Omnikey readers.

iClass is an ISO/IEC 15693 [ISO09] compatible contactless smartcard manufactured by HID Global. It was introduced on the market back in 2002 as a secure replacement of the HID Prox card which had no cryptography at all. According to the manufacturer more than 300 million iClass cards have been sold. These cards are widely used in access control to secured buildings such as The Bank of America Merrill Lynch, the International Airport of Mexico City and the City of Los Angeles among many others[1]. According to HID [Cum06] iClass is also deployed at the United States Navy base of Pearl Harbor. Other applications include secure user authentication such as in the naviGO system included in Dell's Latitude and Precision laptops; e-payment such as in the FreedomPay and SmartCentric systems; and billing of electric vehicle charging such as in the Liberty PlugIns system.

HID Global is also the manufacturer of the popular Omnikey readers. The Omnikey 5321 reader family is a multi-protocol contactless reader which includes iClass compatibility. Starting from firmware version 5.00 these readers have the so-called 'Omnikey Secure Mode' which is required to update iClass card keys. This Secure Mode provides encryption of the USB traffic complying with ISO/IEC 24727 [ISO08] standard.

1.1 Related Work

Experience has shown that, once obscurity has been circumvented, proprietary algorithms often do not provide a satisfactory level of security. One of the most remarkable examples of that is the infamous case of the Mifare Classic [NESP08, GdKGM+08, GvRVS09] used widely in access control and transport ticketing systems. Other examples include KeeLoq [IKD+08] and Hitag2 [SNC09], which are widely used in wireless car keys and the A5/1 [Gol97] and DECT [LST+09] ciphers used in cell and cordless phones.

1.2 Our contribution

The contribution of this paper is manyfold. First it describes the reverse engineering of the built-in key diversification algorithm of iClass. This key diversification algorithm consists of two parts: a cipher that is used to encrypt the identity of the card; and a key fortification function, called hash0 in HID documentation, which is intended to add extra protection to the master key. Our approach for reverse engineering is in line with that of [GdKGM+08, LST+09, GvRVS10] and consists of analyzing the update card key messages sent by an iClass compatible reader while we produce small modifications on the diversified key, just before

[1] http://hidglobal.com/mediacenter.php?cat2=2

fortification. For this it was first necessary to bypass the encryption layer of the Omnikey Secure Mode. We reverse engineered the Omnikey Secure Mode and wrote a library that is capable of communicating in Omnikey Secure Mode to any Omnikey reader. To eavesdrop the contactless interface we have built a custom firmware for the Proxmark III in order to intercept ISO/IEC 15693 [ISO09] frames. We have released the library, firmware and an implementation of hash0 under the GNU General Public License and they are available at the Proxmark website[2].

Last but not least, we show that the key fortification function hash0 is actually not one-way nor collision resistant and therefore it adds little protection to the master key. Concretely, we give the inverse function hash0^{-1} that on input a 64 bit bitstring it outputs a modest amount (on average 4) of candidate pre-images. We propose an attack that recovers a master key from an iClass reader of comparable complexity to that of breaking single DES, thus it can be accomplished within a few days on a RIVYERA[3]. This is extremely sensitive since there is only one master key for all iClass readers and from which all diversified card keys can be computed.

As an alternative, it is possible to emulate a predefined card identity and use a DES rainbow table [Hel80] based on this identity to perform the attack. This allows an adversary to recover the master key within minutes.

During the course of this research, Meriac and Plötz presented a powerful procedure to read out the EEPROM of a PIC microcontroller, like the ones used in iClass readers, at the 27th meeting of the Chaos Communication Congress [MP10, Mer10]. This attack is possible due to a misconfiguration of the memory access control bits of the PIC used in early reader models, for more details on this attack see the OpenPCD website[4]. Their attack on the hardware is a viable alternative to retrieve the master key.

2 Omnikey Secure Mode

The Omnikey contactless smartcard reader has a range of key slots where it stores cryptographic keys. These keys are used to authenticate with an HID iClass card. After a valid authentication the reader gains read and write access to the memory in the card.

All recent Omnikey 5321 and 6321 contactless smartcard readers manufactured by HID Global support encrypted communication with the host, which is called *Secure Mode*. Applications compliant with ISO/IEC 24727 [ISO08] must provide end-to-end encryption and therefore the USB communication between the application and reader needs to be encrypted.

To activate the Secure Mode, the host application uses a 3DES key K_{CUW} to perform mutual authentication with the reader. According to the Omnikey developers guide [WDS+04] this key is only known by a limited group of developers under a non-disclosure agreement with HID Global.

The Omnikey Secure Mode must be active in order to perform security sensitive operations like changing the key of a card. In order to be able to eavesdrop and modify messages between the reader and a card during a key update, the Omnikey Secure Mode must be circumvented.

The two-factor authentication application naviGO from HID Global provides a login procedure for Windows computers using an iClass card and a PIN-code. A trial version of this software package is freely available online[5]. NaviGO uses the Omnikey reader for the personalization phase where it authenticates, updates the key and writes credentials to an iClass card. To perform these actions naviGO needs to know the cryptographic key K_{CUW} in order to use the Secure Mode. HID Global stores the secret key in an unprotected binary file. After extracting K_{CUW} from the file `iCLASSCardLib.dll` we gained full control over the secured USB channel.

We have released a library called *iClassified* that makes it possible to send arbitrary commands to an Omnikey reader using the Omnikey reader in Secure Mode.

3 iClass and PicoPass

The iClass card is basically a re-branded version of the PicoPass contactless smartcard which is manufactured by Inside Secure[6]. The documentation of the PicoPass [Con04] defines the configuration options, commands and memory structure of an iClass 2KS card. Before HID Global sells the PicoPass as an iClass card, they configure the memory, store their cryptographic keys and blow the fuse that allows any future changes to the configuration.

Block	Content	Denoted by
0	Card serial number	Identifier *id*
1	Configuration	
2	e-Purse	Card challenge c_C
3	Key for application 1	Debit key kd_{id}
4	Key for application 2	Credit key kc_{id}
5	Application issuer area	
$6 \ldots 18$	Application 1	HID application a_{HID}
$19 \ldots n$	Application 2	$n = 16x - 1$ for xKS

Figure 1: Memory layout of an iClass card

The iClass cards come in two versions 2KS and 16KS with respectively 256 and 4096 bytes of memory. The memory is divided into blocks of eight bytes as shown in Figure 1. Memory blocks 0, 1, 2 and 5 are publicly accessible, they contain the card serial number *id*, configuration

[2] `http://www.proxmark.org`
[3] `http://www.sciengines.com`
[4] `http://www.openpcd.org/HID_iClass_demystified`
[5] `http://www.hidglobal.com/cardServices/naviGoTrialDownloadForm.php`
[6] `http://www.insidesecure.com/eng/Products/Secure-Solutions/PicoPass`

bits, the card challenge c_C and issuer information. Block 3 and 4 contain two diversified cryptographic keys which are derived from two different HID master keys. These master keys are referred to in the documentation as debit key kd and credit key kc. The card only stores the diversified keys kd_{id} and kc_{id}. The remaining blocks are divided into two areas so-called applications. The size of these applications is defined by the configuration block.

The first application of an iClass card represents the *HID application* which stores the identifier, PIN code, password and other access control information. Read and write access to the HID application requires a valid mutual authentication using a proprietary algorithm that proves knowledge of kd_{id}.

The second application is user defined and secured by a key kc_{id} derived from kc. The default kc (but not kd) is stored in the same binary file that contains the secret key for the Omnikey Secure Mode. We use this key later on Section 4.1 during the reverse engineering process.

We use our *iClassified* library to eavesdrop the USB communication while the card key is updated. We observe that a default iClass master key is loaded into key slot 32 of the reader. This key is used to derive the card key which is used for authentication. Then, a new master key is loaded into slot 32 and the card key is updated with the new derived key. Figure 2 shows the eavesdropped messages between the reader and a card during a sequence of card key update commands. The application first updates the default key kc of an genuine iClass card to random kc' and kc''. Finally it sets the default key again. The trace shows that the key update message contains as payload the exclusive-or (XOR) of the old and new key as mentioned in [MP10]. This can be verified computing $(kc'_{id} \oplus kc_{id}) \oplus (kc''_{id} \oplus kc'_{id}) = kc_{id} \oplus kc''_{id}$.

3.1 Authentication and Key Fortification

This section describes the authentication protocol between an iClass card and reader. Furthermore, it gives an overview of the built-in key diversification algorithm.

The authentication protocol between an iClass card and a reader is depicted in Figure 3. First, the card sends its identity id and a card challenge c_C. This c_C is called 'e-purse' [Con04] and it is special in the sense that it is intended to provide freshness. Apparently, the card lacks a pseudo-random generator and therefore, after a successful authentication, the reader should update c_C to a new value in order to provide freshness in the next authentication. Note that this is not enforced by the card. Next, the reader answers with a nonce n_R of its choosing and an answer a_R to the challenge of the card. This answer is presumable some sort of MAC depending on c_C and n_R. Finally, the card answers with a similar message a_C to achieve mutual authentication.

iClass has a built-in key diversification algorithm. Figure 4 is extracted from the PicoPass datasheet [Con04]. It suggests that the reader encrypts the card identity (id) using single DES. Then it performs a fortification algorithm to obtain the diversified key. The following steps verify that the card identity is the only input to the DES algorithm:

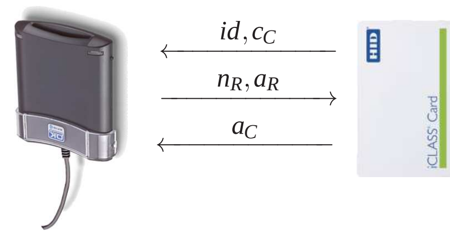

Figure 3: Authentication protocol

- start with any 64 bit bitstring c, e.g., all zeros
- choose a random key k and use DES to decrypt c. This results in a plaintext p
- choose a different key k' and use DES to decrypt c. This results in a plaintext p'
- run a card key update with k with a reader that receives identity p from a card emulator. Repeat this using key k' and identity p' and verify that the derived key k_p is equal to $k'_{p'}$.

Key fortification functions are non-injective functions (many-to-one) which, in contrast with hash functions, intentionally have many collisions [AL94]. The idea behind it is that even if an adversary has access to many diversified keys, these do not univocally determine a master key. This comes, of course, at the cost of loosing entropy in the diversified key.

In practice, it means that even if you manage to invert the fortification function, you will get many candidate preimages which in turn you need to brute force to get to the master secret key.

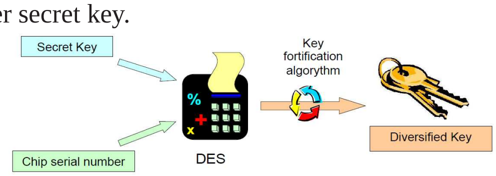

Figure 4: Extracted from the PicoPass datasheet [Con04]

4 Reverse Engineering Key Fortification

This section describes the reverse engineering of the key fortification function. The design of this function, called h0 [Cum03] or hash0 [Cum06], is not publicly available. Our primary goal is to learn the card key derivation which gives complete control over the card key. In order to reach this goal it is necessary to reverse engineer the fortification function.

As explained in Section 3.1 the input to the key diversification is a master secret key (e.g., kc or kd) and a card identity id. From this key, say kc, and id a ciphertext $c = \mathrm{DES_{enc}}(id, kc)$ is computed. Finally, the actual diversified key kc_{id} is computed $\mathrm{hash0}(c) = kc_{id}$.

Origin	Message	Description
Reader	`0c 00 73 33`	Read identifier
Tag	`86 1d c1 00 f7 ff 12 e0`	Card serial number *id*
Reader	`0c 01 fa 22`	Read configuration
Tag	`12 ff ff ff 7f 1f ff 3c`	iClass 2KS configuration
Reader	`18 02`	Authenticate with kc_{id}
Tag	`fe ff ff ff ff ff ff ff`	Card challenge c_C
Reader	`05 00 00 c1 d9 7e 99 bb f4`	Reader challenge (`05`, n_R, a_R)
Tag	`46 3c 62 98`	Response (a_C)
Reader	`87 04 fc b4 32 3e 6a 86 56 26 8a b5 18 cc`	Update kc_{id} (`87 04`, $kc'_{id} \oplus kc_{id}$, `8a b5 18 cc`)
Tag	`ff ff ff ff ff ff ff ff`	Update succesful
Reader	`0c 00 73 33` ...	Read *id*
Reader	`87 04 76 98 db 5d 01 78 0a 8f 67 25 c1 08` ...	Update kc_{id} (`87 04`, $kc''_{id} \oplus kc'_{id}$, `67 25 c1 08`)
Reader	`87 04 8a 2c e9 63 6b fe 5c a9 e2 a5 bc 55`	Update kc_{id} (`87 04`, $kc_{id} \oplus kc''_{id}$, `e2 a5 bc 55`)

Figure 2: Authenticate and update keys of an iClass card

4.1 Input-Output Relations

A good first step to recover hash0 is to analyze its input-output relations on bit level. This requires complete control over its input c which can be achieved in a test setup by the emulation of a card identity *id* knowing the master key *kc*.

The following steps deliver XOR differences between two hash0 evaluations that differ only one bit in the input:

- generate a large set of random bitstrings $c_i \in \{0,1\}^{64}$.
- for each c_i calculate $id_i = \mathrm{DES_{dec}}(c_i, kc)$ and $id_i^j = \mathrm{DES_{dec}}(c_i \oplus 2^j, kc)$ for $j \in \{0,\dots,63\}$.
- for each c_i execute 64 key updates as follows:
  - authenticate with id_i
  - perform a key update, the reader requests the card identity again, now use id_i^j instead of id_i

Keep the key *kc* constant during the key updates described above. This delivers the XOR of two function evaluations of the form $\mathrm{hash0}(c_i) \oplus \mathrm{hash0}(c_i \oplus 2^j)$. We performed this procedure for 3000 values c_i with $j \in \{0,\dots,63\}$. The results are grouped by the position of the flipped bit. Then, the AND and OR is computed of all the results in a group. These cumulative AND and OR-masks for 64 bitflips in 3000 random bitstrings c_i are presented in Figure 6 and 9.

4.2 Function Input Partitioning

Figure 6 shows that the hash0 function handles the 48 rightmost bits in smaller 6-bit pieces. These 6-bit data chunks are defined as $z_0,\dots,z_7$. The two bytes on the left are defined x and y. Here x defines a permutation on the output and the individual bits of y define whether or not a complement operation is applied on one of the 6-bit output values. The eight output bytes are defined as $k_0,\dots,k_7$ and constitute the diversified key kc_{id}. Similarly, the input c to the hash0 function is constituted by $c = \langle x, y, z_0, \dots, z_7 \rangle$.

For the ease of reading we write $x_{[b]}$ to denote the b-th bit of variable x where $x_{[0]}$ means the rightmost bit of x.

The structure of the masks in Figure 6 and 9 are computed with $x = y = 0$ and $z_0,\dots,z_7$ as random bitstrings. The masks lead to the following observations:

- $z_0,\dots,z_3$ affects $k_4,\dots,k_7$.
- $z_4,\dots,z_7$ affects $k_0,\dots,k_3$.
- $z_0,\dots,z_3$ and $z_4,\dots,z_7$ generate a similar structure in the output but are mutually independent. This suggests that there is a subfunction that is called twice, once with $z_0,\dots,z_3$ and once with $z_4,\dots,z_7$. In the context of this paper we refer to this function as scramble.
- $y_{[i]}$ affects k_i for $i \in \{0,\dots,7\}$. The OR-mask for y indicates a complement operation on the output while the AND-mask presumes an injective function that maps $y_{[i]}$ to $k_{i[7]}$.
- x creates a permutation. The output is scrambled after flipping a single bit within x. The AND-mask shows that $k_{i[0]}$ is exclusively affected by x for $i \in \{0,\dots,7\}$.
- flipping bits in $z_0,\dots,z_7$ does never affect the left- or rightmost bits of $k_0,\dots,k_7$. This is inferred from the occurrences of the `0x7e` value in the OR-mask which is `01111110` in binary.

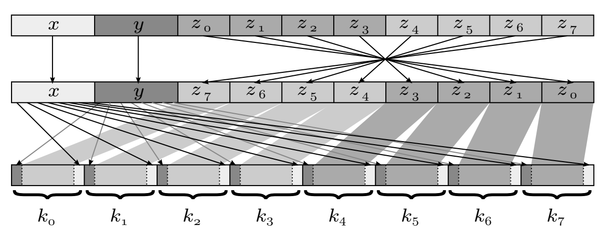

Figure 5: Partitioned Function Input for $x = 0$

The above observations suggest that the problem of function recovery can be split into parts. Figure 5 summarizes how different parts of the input affect specific parts of the output when x is kept zero. Note that the last observation shows that the subfunction scramble operates on four 6-bit input values and computes four 6-bit output values. These output values constitute the middle 6 bits of output bytes k_i, see Figure 5. Furthermore, it is observed that the ordering of the 6-bit output values and the leftmost bit of the output bytes are determined by x. Each bit of y is simply copied into the rightmost bit of each output byte.

Summarizing, the hash0 function can be split into three different parts. The first part is the subfunction scramble which is called twice, once with input $z_0,\ldots,z_3$ and once with input $z_4,\ldots,z_7$. The second part computes a bitwise complement operation based on the *complement* byte y and the last part applies a permutation that is defined by the *permute* byte x. The following sections discuss these different parts of the hash0 function. Finally, Section 4.6 defines the complete function.

	bit	OR-mask	AND-mask
	$\oplus \rightarrow$	$k_0k_1k_2k_3k_4k_5k_6k_7$	$k_0k_1k_2k_3k_4k_5k_6k_7$
z_7	0	**7e7e7e7e**00000000	**04**00000000000000
	1	**7e7e7e7e**00000000	**04**00000000000000
	2	**7a7e7e7e**00000000	**08**00000000000000
	3	**727e7e7e**00000000	**10**00000000000000
	4	**627e7e7e**00000000	**20**00000000000000
	5	**427e7e7e**00000000	**40**00000000000000
z_6	6	00**7e7e7e**00000000	0000000000000000
		…	…
	11	00**7e7e7e**00000000	0000000000000000
z_5	12	0000**7e7e**00000000	0000000000000000
		…	…
	17	0000**7e7e**00000000	0000000000000000
z_4	18	000000**7e**00000000	0000000000000000
		…	…
	23	000000**7e**00000000	0000000000000000
z_3	24	00000000**027e7e7e**	00000000**02**000000
	25	00000000**047e7e7e**	00000000**04**000000
	26	00000000**087e7e7e**	00000000**08**000000
	27	00000000**107e7e7e**	00000000**10**000000
	28	00000000**207e7e7e**	00000000**20**000000
	29	00000000**407e7e7e**	00000000**40**000000
z_2	30	0000000000**7e7e7e**	0000000000000000
		…	…
	35	0000000000**7e7e7e**	0000000000000000
z_1	36	000000000000**7e7e**	0000000000000000
		…	…
	41	000000000000**7e7e**	0000000000000000
z_0	42	00000000000000**7e**	0000000000000000
		…	…
	47	00000000000000**7e**	0000000000000000

Figure 6: OR and AND-mask for bitflip 0-47

4.3 Subfunction scramble

This section describes the reverse engineering of the subfunction scramble which operates on four 6-bit input values $z_0,\ldots,z_3$. In order to recover this part of the function we keep $x = y = 0$ while $z_0,\ldots,z_7$ are randomly chosen. For the scramble subfunction only bitflips at positions 0 to 47 matter (see Figure 6). It makes sense to start with the recovery of either k_0 or k_4 as they both depend on a single input z_i. Notice that k_4 is just z_3 shifted one bit to the left since we keep $x = y = 0$. However, k_0 seems less predictable. The XOR between two outputs $k_i \oplus k_i'$ of two function calls is defined as $k_i^{\oplus}$. Furthermore, be aware that the subfunction scramble only affects bits $k_{i[1]},\ldots,k_{i[6]}$ (See Fig 5). To put it differently, the output is *always* shifted one bit to the left and therefore this shift can be omitted from the analysis.

In order to find a relation between input values z_7 and output values $k_0^{\oplus}$ a selection of all observed values $k_0^{\oplus}$ is made. Figure 7 shows a relation between z_7 and $k_0^{\oplus}$ and shows which bits of z_7 are fixed for a certain output value $k_0^{\oplus}$. Bits that do not matter are marked with a dot and the bitflip is marked **f**. The two inputs are z_7 where **f** = 0 and z_7' where **f** = 1.

z_7/z_7'	$k_0^{\oplus}$	z_7/z_7'	$k_0^{\oplus}$
....0**f**	06	....**f**0	04
...01**f**	0e	...0**f**1	0c
..011**f**	1e	..01**f**1	1c
.0111**f**	3e	.011**f**1	3c
11111**f**	7c	0111**f**1	7c
01111**f**	7e	1111**f**1	7e

Figure 7: Input-output relations for $k_0^{\oplus}$

The relation is represented for every two inputs z_7 and z_7' as $k_{0[1..6]}^{\oplus} = (z_7 \bmod 63) + 1 \oplus (z_7' \bmod 63) + 1$ which gives confidence that $k_{0[1..6]} = (z_7 \bmod 63) + 1$. The next step is to find $k_{1[1..6]}$ which is dependent on two input input values, namely z_6 and z_7. Again, an overview of all input-output relations (Figure 8) is constructed. The first part where $k_1^{\oplus} \in \{02, 0c, 52, 6c, \ldots\}$ is the result of flipping $z_{6[0]}$ and the second part where $k_1^{\oplus} \in \{0c, 1c, 3c, \ldots, 4e, 64, \ldots\}$ is the result of flipping $z_{6[1]}$.

The observations for flipping $z_{6[0]}$ and $z_{6[1]}$ show that in 97 % of the cases input z_6 and z_7 are independent. 3 % of the bitflips in z_6 make $z_6 + 1$ equal to z_7 or destroy this equality instead.

%	z_6/z_6'	z_7	$k_1^{\oplus}$	
0.97	.....**f**	......	02	bitflip $z_{6[0]}$
0.03	00010**f**	000101	0c	
	10011**f**	101000	52	
	11001**f**	110100	6c	
	.....**f**	......	…	
0.97	....**f**.	......	0c	bitflip $z_{6[1]}$
	...1**f**.	......	1c	
	.011**f**.	......	3c	
	1111**f**.	......	78	
	0111**f**.	......	7c	
0.03	0010**f**0	001001	1a	
	0110**f**0	011001	3a	
	1001**f**0	100111	4e	
	1100**f**1	110100	64	
	....**f**.	......	…	

Figure 8: Input-output relations for $k_1^{\oplus}$

When $z_{6[1]}$ is flipped more output variations in $k_1^{\oplus}$ are observed. Example for $k_1^{\oplus} = \texttt{0x3c}$:

$$\begin{array}{lll} z_6 = \texttt{001101}, & z_6 + 2 = & \texttt{.001111.} \\ z_6' = \texttt{001111}, & z_6' + 2 = & \texttt{.010001.} \quad \oplus \\ \hline & & \texttt{00111100} = \texttt{0x3c} \end{array}$$

The result $k_1^{\oplus} = \texttt{78}$ comes from a modulo operation. Here input z_6 is taken modulo 62, which is $\texttt{111110}$ in binary. Example for $k_1^{\oplus} = \texttt{0x78}$:

$$
\begin{array}{lll}
z_6 = \texttt{111100}, \quad (z_6 \bmod 62)+2 & = & \texttt{.111110.} \\
z_6' = \texttt{111110}, \quad (z_6' \bmod 62)+2 & = & \texttt{.000010.} \quad \oplus \\
\hline
 & & \texttt{01111000} = \texttt{0x78}
\end{array}
$$

Then, 3 % of the output variations invoked by bitflips in $z_{6[1]}$ describe a relation $z_6 + 1 = z_7$. The corresponding $k_1^{\oplus}$ is obtained by taking $k_{1[1..6]} = 1$ when the relation holds and $k_{1[1..6]} = (z_6 \bmod 62) + 2$ when it does not hold. Example for $k_1^{\oplus} = \texttt{0x4e}$:

$$
\begin{array}{lll}
z_6 = \texttt{100100}, \ (z_6 \bmod 62)+2 & = & \texttt{.100110.} \\
z_6' = \texttt{100110}, \ ((z_6' \bmod 62)+1 = n_7) & = & \texttt{.000001.} \quad \oplus \\
\hline
 & & \texttt{01001110} = \texttt{0x4e}
\end{array}
$$

Eventually, the function for $k_{1[1..6]}$ is:

$$
k_{1[1..6]} = \begin{cases} 1, & (z_6 \bmod 62) + 1 = (z_7 \bmod 63); \\ (z_6 \bmod 62) + 2, & \textit{otherwise.} \end{cases}
$$

The remaining $k_{2[1..6]}$ and $k_{3[1..6]}$ can be found in a similar way by flipping bits in the input and closely looking at the input-output relations. Also, it helps to look for related modulo operations on z_5 and z_4. We give $k_{2[1..6]}$ to give some idea of the evolving structure of the function:

$$
k_{2[1..6]} = \begin{cases} 2, & (z_5 \bmod 61) + 1 = (z_6 \bmod 62); \\ & \quad \wedge (z_7 \bmod 63) \neq 0; \\ 1, & (z_5 \bmod 61) + 1 = (z_6 \bmod 62) \\ & \quad \wedge (z_7 \bmod 63) = 0; \\ 1, & (z_5 \bmod 61) + 2 = (z_7 \bmod 63); \\ (z_5 \bmod 61) + 3, & \textit{otherwise.} \end{cases}
$$

After the recovery of the first block $z_4, \ldots, z_7$ it is relatively easy to find the subfunction for $z_0, \ldots, z_3$. The modulos and additions differ but the structure of the function is completely the same. For this reason it is possible to write it as a subfunction scramble that is called twice, once for $z_0, \ldots, z_3$ and once for $z_4, \ldots, z_7$. The final subfunction scramble is given by Definition 4.1.

4.4 Complement Byte

The complement byte y performs a complement operation on the output of the function. Figure 9 shows that flipping a bit $y_{[i]}$ means that bit $k_{i[7]}$ is flipped for $i \in \{0, \ldots, 7\}$. Notice that no other input bit influences any $k_{i[7]}$. Furthermore, $k_{i[1]}, \ldots, k_{i[6]}$ are flipped but be aware that these bits might come from any other z_j due to the permute byte x. Finally, every $k_{i[0]}$ is not affected. It is important to observe that for $k_4, \ldots, k_7$ the OR and AND-mask agree that the left 7 bits are always flipped while for $k_0, \ldots, k_3$ this is not true. To be precise, the bits $k_{0[1]}, k_{1[1]}, k_{2[1]}$ and $k_{3[1]}$ are *never* flipped. This is because the 6-bit output value z_j that constitutes output byte k_i is decremented by one if $j \leq 3$ except when $y_{[i]} = 0$. Example for $k_0^{\oplus} = \texttt{0xfc}$:

$$
\begin{array}{lll}
z_j = \texttt{101101}, \quad \text{where } j \leq 3 & & \\
y_0 = \mathbf{0}, \quad k_0 = y_0 \cdot z_j \cdot t & = & \texttt{0101101}t \\
y_0' = \mathbf{1}, \quad k_0' = y_0' \cdot \overline{z_j - 1} \cdot t & = & \texttt{1010011}t \quad \oplus \\
\hline
 & & \texttt{11111100} = \texttt{0xfc}
\end{array}
$$

	bit	OR-mask	AND-mask
	$\oplus \rightarrow$	$k_0k_1k_2k_3k_4k_5k_6k_7$	$k_0k_1k_2k_3k_4k_5k_6k_7$
y	48	fc00000000000000	8000000000000000
	49	00fc000000000000	0080000000000000
	50	0000fc0000000000	0000800000000000
	51	000000fc00000000	0000008000000000
	52	00000000fe000000	00000000fe000000
	53	0000000000fe0000	0000000000fe0000
	54	000000000000fe00	000000000000fe00
	55	00000000000000fe	00000000000000fe
x	56	7f7f7f7e7e7f7f7f	0101010000010101
	57	00007f7e7f000000	0000010001000000
	58	7f7e7e7e7f000000	0100000001000000
	59	7f7e7e7e7e7f0000	0100000000010000
	60	00007f7e7e7e7f00	0000010000000100
	61	7f7e7f7f7f7f7f00	0100010101010100
	62	7f7e7f7e7e7f7f00	0100010000010100
	63	7f7e7f7e7f7e7f00	0100010001000100

Figure 9: OR and AND-mask for bitflip 48-63

```
π = [
01234567, 35670124, 01342567, 15670234, 12340567,
34670125, 01352467, 14670235, 12350467, 23670145,
02451367, 12670345, 12450367, 02671345, 23450167,
34570126, 01362457, 14570236, 12360457, 23570146,
02461357, 03571246, 03461257, 02571346, 23460157,
23470156, 02561347, 03471256, 03561247, 02471356,
23560147, 12370456, 14560237, 01372456, 34560127,
45670123, 01243567, 25670134, 02341567, 05671234,
01253467, 24670135, 02351467, 04671235, 01452367,
13670245, 03451267, 03671245, 13450267, 01672345,
01263457, 24570136, 02361457, 04571236, 01462357,
13570246, 12460357, 12570346, 13460257, 01572346,
01562347, 13470256, 12560347, 12470356, 13560247,
01472356, 04561237, 02371456, 24560137, 01273456 ]
```

Figure 10: Permutation π

4.5 Permute Byte

Finally, byte x applies a permutation. Iterating over x while keeping y and $z_0, \ldots, z_7$ constant shows that x is taken modulo 70 since the same output is repeated again for every 70 consecutive inputs. The cumulative bitmasks of the output differences, shown in Figure 9, do not give direct information about this permutation but do make clear that $k_{i[0]}$ is affected. Experiments show that x is an injective mapping on $k_{i[0]}$ for $i = 0, \ldots, 7$. This means that it is possible to learn x from $k_{i[0]}$. Furthermore, the permutation is independent of y and z_i. This means that a table of mappings can be constructed which takes x as index and has particular mappings as its entries. The mappings are presented in Figure 10. To illustrate, $\pi_0 = 01234567$ means that there is no mixing at all and $\pi_2 = 01342567$ means that k_0 stays at position 0 while k_4 is moved to position 2. To isolate one particular mapping we write $\pi_x(i)$ which returns the target position of 6-bit output value $\hat{z}_i$.

4.6 Diversification and Fortification

This section describes the recovered key diversification and fortification procedure. Definition 4.2 gives the definition of the function hash0. It uses a subfunction scramble which is defined by Definition 4.1. First, the key diversification procedure where a diversified key kc_{id} is computed from a card identity id and master key kc is as follows:

$$kc_{id} = \text{hash0}(\text{DES}_{\text{enc}}(id, kc))$$

Here the DES encryption of id with master key kc outputs a cryptogram c of 64 bits. These 64 bits are divided as $c = \langle x, y, z_0, \dots, z_7 \rangle \in \mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8$ and used as input to the hash0 function. Finally, the output of the hash0 function is $kc_{id} = \langle k_0, \dots, k_7 \rangle \in (\mathbb{F}_2^8)^8$.

The function hash0 first computes $x' = x \bmod 70$ which results in 70 possible permutations (See Fig. 10). Then for all z_i the modulus and additions are computed before calling the subfunction scramble.

Then, the subfunction scramble is called twice, first on input $z'_0, \dots, z'_3$ and then on input $z'_4, \dots, z'_7$. The definition of the function scramble is as follows.

Definition 4.1. *Let the function scramble*: $(\mathbb{F}_2^6)^4 \to (\mathbb{F}_2^6)^4$ *be defined as*

$$scramble(z_0 \dots z_3) = sc(0, 1, z_0 \dots z_3)$$

where $sc\colon \mathbb{N} \times \mathbb{N} \times (\mathbb{F}_2^6)^4 \to (\mathbb{F}_2^6)^4$ *is defined as*

$$sc(2, 4, z_0 \dots z_3) = z_0 \dots z_3$$
$$sc(i, 4, z_0 \dots z_3) = sc(i+1, i+2, z_0 \dots z_3)$$
$$sc(i, j, z_0 \dots z_3) =$$
$$\begin{cases} sc(i, j+1, z_0..z_i \leftarrow (3-j)..z_3), & z_i = z_j; \\ sc(i, j+1, z_0 \dots z_3), & \textit{otherwise.} \end{cases}$$

After this a permutation is applied to the output bytes. The definition of hash0 is as follows.

Definition 4.2. *Let the function hash0*: $\mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8 \to (\mathbb{F}_2^8)^8$ *be defined as*

$$hash0(x, y, z_0 \dots z_7) = k_0 \dots k_7$$

where

$$x' = x \mod 70$$
$$z'_i = (z_i \mod 61 + i) + 3 - i \qquad i = 0 \dots 3$$
$$z'_i = (z_i \mod 56 + i) + 7 - i \qquad i = 4 \dots 7$$
$$\hat{z}_0 \dots \hat{z}_3 = scramble(z'_0 \dots z'_3)$$
$$\dot{z}_4 \dots \dot{z}_7 = scramble(z'_4 \dots z'_7)$$
$$\hat{z}_i = \dot{z}_i + \overline{y_{[\pi_{x'}(7-i)]}} \qquad i = 4 \dots 7$$

$$k_{\pi_{x'}(i)} = \begin{cases} y_{[\pi_{x'}(i)]} \cdot \hat{z}_{7-i} \cdot (i > 3), & y_{[\pi_{x'}(i)]} = 0; \\ y_{[\pi_{x'}(i)]} \cdot \overline{\hat{z}_{7-i}} \cdot (i > 3), & \textit{otherwise.} \end{cases}$$

$$i = 0 \dots 7$$

5 Weaknesses

This section describes weaknesses in the design of the Omnikey Secure Mode and on the iClass built-in key diversification and fortification algorithms. These weaknesses will be later exploited in Section 6.

5.1 Omnikey Secure Mode

Even though encrypting the communication over USB is in principle a good practice, the way it is implemented in the Omnikey Secure Mode adds very little security. The shared key k_{CUW} is the same for all Omnikey readers and it is included in software that is publicly available online. This only gives a false feeling of added security.

5.2 Weak key diversification algorithm

iClass uses single DES encryption for key diversification. This provides very weak protection of the master key. This is a critical weakness, especially considering that there is only one master key for the HID application for all iClass cards.

The manufacturer seems to be aware of this weakness and tries to tackle the problem by adding the key fortification function.

This comes at the price of loosing entropy on the diversified card keys. After the DES computation the diversified 64-bit card key have at most 56 bit of entropy. Then, this key is put through the fortification function where it looses another 2.2 bits of entropy. In the next section, we explain where these 2.2 bits come from and discuss the security properties of the fortification function.

5.3 Weak key fortification

This section clarifies why the key fortification is not strengthening the diversified key kc_{id}. First, note that only the modulo operations in hash0 on x ($\frac{256}{70}$) and $z_0, \dots, z_2, z_4, \dots, z_7$ are responsible for the collisions in the output. The expected number of pre-images for an output of hash0 is given by:

$$\frac{256}{70} \times \frac{64}{60} \times \prod_{n=61}^{63} \left(\frac{64}{n}\right)^2 \approx 4.72$$

These modulo operations make inverting the function straightforward. For every pre-image one needs to determine if there exists another value within the input domain that leads to the same output when the modulus is taken. Note that each input value z_i may have a second pre-image that maps to the same output value. Furthermore, every permute byte x has at least two other values that map to the same output value and in some cases there is even a third one. This means that the minimal number of pre-images is three. The probability p that for a given random input c there are only two other pre-images is:

$$p = \frac{24}{70} \times \frac{60}{64} \times \prod_{n=61}^{63} \left(\frac{n}{64}\right)^2 \approx 0.27$$

This means that hash0 does not add that much of additional protection. For example, imagine an attacker who can learn the output kc_{id} of $\text{hash0}(\text{DES}_{\text{enc}}(id,kc))$ for arbitrary values id. Then, the probability p' for an attacker to obtain an output kc_{id} which has only three pre-images is $p' = 1-(1-p)^n$, where n is the number of function calls using random identities id. For $n = 15$ this probability $p' > 0.99$.

5.4 Inverting hash0

It is relatively easy to compute the inverse of the function hash0. Let us first compute the inverse of the function scramble. Observe that the function scramble^{-1} is defined just as scramble except for one case where the condition and assignment are swapped. Concretely,

Definition 5.1. *Let the function* $\mathit{scramble}^{-1}\colon (\mathbb{F}_2^6)^4 \to (\mathbb{F}_2^6)^4$ *be defined just as* $\mathit{scramble}(z_0 \dots z_3)$ *except for the following case where*

$$sc^{-1}(i,j,z_0\dots z_3) = \begin{cases} sc^{-1}(i,j+1,z_0..z_i \leftarrow z_j..z_3), & z_i = 3-j; \\ sc^{-1}(i,j+1,z_0\dots z_3), & \textit{otherwise.} \end{cases}$$

Next, we define the function hash0^{-1}, the inverse of hash0. This function outputs a set $\mathscr{C}$ of candidate pre-images. These pre-images output the same key k when applying hash0. The definition of hash0^{-1} is as follows.

Definition 5.2. *Let the function* $\mathit{hash0}^{-1}\colon (\mathbb{F}_2^8)^8 \to \{\mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8\}$ *be defined as*

$$\mathit{hash0}^{-1}(k_0\dots k_7) = \mathscr{C}$$

where

$$\begin{aligned} \mathscr{C} = &\{x \mid x \equiv x' \mod 70\} \times \{y\} \times \\ &\{z_0 \mid z_0 \equiv \tilde{z}_0 \mod 61\} \times \{z_1 \mid z_1 \equiv \tilde{z}_1 \mod 62\} \times \\ &\{z_2 \mid z_2 \equiv \tilde{z}_2 \mod 63\} \times \{z_3 \mid z_3 \equiv \tilde{z}_3 \mod 64\} \times \\ &\{z_4 \mid z_4 \equiv \tilde{z}_4 \mod 60\} \times \{z_5 \mid z_5 \equiv \tilde{z}_5 \mod 61\} \times \\ &\{z_6 \mid z_6 \equiv \tilde{z}_6 \mod 62\} \times \{z_7 \mid z_7 \equiv \tilde{z}_7 \mod 63\} \end{aligned}$$

x' is the unique element in $\mathbb{F}_2^8$ s.t. $(\pi_{x'}(i) > 3) \Leftrightarrow (k_{i[7]} = 1)$, for $i = 0\dots 7$.

$$y_{[i]} = k_{\pi_{x'}(i)[0]} \qquad i = 0\dots 7$$

$$\tilde{z}_i = z'_i - (3-(i \mod 4)) \qquad i = 0\dots 7$$

$$z'_0\dots z'_3 = \mathit{scramble}^{-1}(\hat{z}_0\dots\hat{z}_3)$$

$$z'_4\dots z'_7 = \mathit{scramble}^{-1}(\dot{z}_4\dots\dot{z}_7)$$

$$\dot{z}_{[i]} = \hat{z}_{[i]} - y_{[\pi_{x'}(7-i)]} \qquad i = 4\dots 7$$

$$\hat{z}_i = \begin{cases} k_{\pi_{x'}(7-i)[1\dots 6]}, & y_{[\pi_{x'}(7-i)]} = 0; \\ \overline{k_{\pi_{x'}(7-i)[1\dots 6]}}, & \textit{otherwise.} \end{cases} \qquad i = 0\dots 7$$

6 Key recovery attack

From the weaknesses that were explained in the previous section it can be concluded that hash0 does not significantly increase the complexity of an attack on the master key kc. In fact, the attack explained in this section requires one brute force run on DES. For this key recovery attack an attacker needs to control a reader and be able to issue key update commands. This is the case, for example, in the Omnikey Secure Mode. The attack consists of two phases:

Phase 1

- emulate a random identity id to the reader
- issue an update key command that updates from a known user defined key kc' to the unknown master key kc. Now, $id_{kc} = \text{hash0}(\text{DES}_{\text{enc}}(id,kc))$ can be obtained from the XOR difference.
- compute the pre-images c_i of id_{kc}.
- repeat Phase 1 until an output id_{kc} is obtained which has three pre-images.

Phase 2

- for every candidate key $kt \in \{0,1\}^{56}$ check if $\text{DES}_{\text{enc}}(id,kt) = c_i$ for $i \in \{0,1,2\}$
- when the check above succeeds the corresponding key kt needs to be verified against another set of id and kc_{id}.

We verified this attack on the two master keys kc and kd that are stored in the Omnikey reader for the iClass application. The first key kc was also stored in the naviGO software and we could check the key against pre-images that were selected as described above. Although we did not find kd stored in software we were still able to verify it since we could dump the EEPROM of a reader where kd was stored.

The attack above comes down to a brute force attack on single DES. A slightly different variant is to keep the card identity id fixed and use a DES rainbow table [Hel80] that is constructed for a specific plaintext and runs through all possible encryptions of this plaintext. Note that the rainbow table needs to be pre-computed and thus a fixed plaintext is chosen on forehand. This means that one fixed predefined id is to be used in the attack. The number of pre-images can no longer be controlled. In the worst case the number of pre-images is 512.

7 Conclusions

In this paper we have shown that obscurity does not provide extra security and it can be circumvented. In fact, experience shows that instead of adding extra security it often covers for negligent designs.

It is hard to imagine why HID decided, back in 2002, to use single DES for key diversification considering that DES was already broken in practice in 1997 [Fou98]. Especially when most (if not all) HID readers are capable of computing 3DES. Another unfortunate choice was to design their proprietary hash0 function instead of using an openly designed and community reviewed hash function like SHA-1. From a cryptographic perspective, their proprietary function hash0 fails to achieve any desirable security goal.

References

[AL94] RJ Anderson and TMA Lomas. Fortifying key negotiation schemes with poorly chosen passwords. *Electronics letters*, 30(13):1040–1041, 1994.

[Con04] Inside Contactless. Datasheet PicoPass 2KS. Technical report, November 2004.

[COQ09] Nicolas T. Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical Algebraic Attacks on the Hitag2 Stream Cipher. In *Information Security*, volume 5735 of *Lecture Notes in Computer Science*, pages 167–176. Springer, 2009.

[Cum03] Nathan Cummings. iCLASS Levels of Security. Technical report, April 2003.

[Cum06] Nathan Cummings. Sales Training. Presentation Slides from HID Technologies, 2006.

[Fou98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[GdKGM+08] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling Mifare Classic. In *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2008.

[Gol97] Jovan Dj. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *EUROCRYPT 1997*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255, 1997.

[GvRVS09] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a Mifare Classic card. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 3–15. IEEE, 2009.

[GvRVS10] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Dismantling SecureMemory, CryptoMemory and CryptoRF. In *17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 250–259. ACM, 2010.

[Hel80] M. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.

[IKD+08] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelmann, Eli Biham, and Bart Preneel. A Practical Attack on KeeLoq. In *Advances in Cryptology - EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2008.

[ISO08] ISO/IEC. 24727 - Identification Cards – Integrated Circuit Card Programming Interfaces. Technical report, 2008.

[ISO09] ISO/IEC. 15693 - Identification cards – Contactless integrated circuit cards – Vicinity cards. Technical report, 2009.

[LST+09] S. Lucks, A. Schuler, E. Tews, R.P. Weinmann, and M. Wenzel. Attacks on the DECT authentication mechanisms. *Topics in Cryptology–CT-RSA 2009*, pages 48–65, 2009.

[Mer10] Milosch Meriac. Heart of darkness - exploring the uncharted backwaters of hid iclass security. `http://www.openpcd.org/images/HID-iCLASS-security.pdf`, 2010.

[MP10] Milosch Meriac and Henryk Plötz. Analyzing a modern cryptographic RFID system HID iClass demystified. Presentation at the 27th Chaos Computer Congress, December 2010.

[NESP08] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse Engineering a Cryptographic RFID Tag. In *USENIX Security '08*, pages 185–193, 2008.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer Berlin / Heidelberg, 2009.

[WDS+04] Werner Waitz, L Dixon, S Schwab, L Hanna, T Muth, Marc Jacquinot, and Abu Ismail. OMNIKEY Contactless Smart Card Readers Developers Guide. Technical report, November 2004.