Wirelessly Lockpicking a Smart Card Reader

Flavio D. Garcia
School of Computer Science
University of Birmingham, UK
f.garcia@bham.ac.uk

Gerhard de Koning Gans *Roel Verdult*
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands.
{gkoningg,rverdult}@cs.ru.nl

Abstract

With more than 300 million cards sold, HID iClass is one of the most popular contactless smart cards on the market. It is widely used for access control, secure login and payment systems. The card uses 64-bit keys to provide authenticity and integrity. The cipher and key diversification algorithms used in iClass are proprietary and little information about them is publicly available. In this paper we have reverse engineered all security mechanisms in the card including cipher, authentication protocol and also key diversification algorithms, which we publish in full detail. Furthermore, we have found six critical weaknesses that we exploit in two attacks, one against iClass Standard and one against iClass Elite (a.k.a., iClass High Security). In order to recover a secret card key, the first attack requires one authentication attempt with a legitimate reader and 2^{22} queries to a card. This attack has a computational complexity of 2^{40} MAC computations. The whole attack can be executed within a day on ordinary hardware. Remarkably, the second attack which is against iClass Elite is significantly faster. It directly recovers the system wide master key from only 15 authentication attempts with a legitimate reader. The computational complexity of this attack is lower than 2^{25} MAC computations, which means that it can be fully executed within 5 seconds on an ordinary laptop.

1. Introduction

iClass is an ISO/IEC 15693 [ISO00], [ISO06], [ISO09] compatible contactless smart card manufactured by HID Global. It was introduced in the market back in 2002 as a secure replacement of the HID Prox card which did not have any cryptographic capabilities. The iClass cards are widely used in access control of secured buildings such as The Bank of America Merrill Lynch, the International Airport of Mexico City and the United States Navy base of Pearl Harbor [Cum06] among many others (see http://hidglobal.com/mediacenter.php?cat2=2). Other applications include secure user authentication such as in the naviGO system included in Dell's Latitude and Precision laptops; e-payment like in the FreedomPay and SmartCentric systems; and billing of electric vehicle charging such as in the Liberty PlugIns system. iClass has also been incorporated into the new BlackBerry phones which support Near Field Communication (NFC). iClass uses a proprietary cipher to provide data integrity and mutual authentication between card and reader. The cipher uses a 64-bit diversified key which is derived from a 56-bit master key and the serial number of the card. This key diversification algorithm is built into all iClass readers. The technology used in the card is covered by US Patent 6058481 and EP 0890157. The precise description of both the cipher and the key diversification algorithms are kept secret by the manufacturer following the principles of security by obscurity. HID distinguishes two system configurations for iClass, namely iClass Standard and iClass Elite. The main differences between iClass Standard and iClass Elite lies in their key management and key diversification algorithms. Remarkably, all iClass Standard cards worldwide share the same master key for the iClass application. This master key is stored in the EEPROM memory of every iClass reader. Our analysis uncovers this key. In iClass Elite, however, it is possible to let HID generate and manage a custom key for your system if you are willing to pay a higher price. The iClass Elite Program (a.k.a., High Security) uses an additional key diversification algorithm (on top of the iClass Standard key diversification) and a custom master key per system which according to HID provides "the highest level of security" [HID09].

2. Research context and related work

Over the last few years, much attention has been paid to the (in)security of the cryptographic mechanisms used in contactless smart cards [GdKGM+08], [GvRVWS10], [PN12], [VGB12]. Experience has shown that the secrecy of proprietary ciphers does not contribute to their cryptographic strength. Most notably the

Mifare Classic, which has been thoroughly broken in the last few years [NESP08], [dKGHG08], [GdKGM+08], [GvRVWS09], [Cou09]. Other prominent examples include KeeLoq [Bog07], [KKMP09], Megamos [VGE13] and Hitag2 [COQ09], [SNC09], [vN11], [SHXZ11], [VGB12] used in car keys, CryptoRF [GvRVWS10], [BKZ11], [BGV+12] used in access control and payment systems and the A5/1 [Gol97], DECT [LST+09] and GMR [DHW+12] ciphers used in cordless/GSM phones. HID proposes iClass as a migration option for systems using Mifare Classic, boosting that iClass provides "improved security, performance and data integrity"[1]. The details of the security mechanisms of iClass remained secret for almost one decade.

During the course of our research Kim, Jung, Lee, Jung and Han have made a technical report [KJL+11] available online describing independent reverse engineering of the cipher used in iClass. Their research takes a very different, hardware oriented approach. They recovered most of the cipher by slicing the chip and analyzing the circuits with a microscope. Our approach, however, is radically different as our reverse engineering is based on the disassembly of the reader's firmware and the study of the communication behavior of tags and readers. Furthermore, the description of the cipher by Kim et al. contains a major flaw. Concretely, their key byte selection function in the cipher is different from the one used in iClass which results in incompatible keys. Kim et al. have proposed two key recovery attacks. The first one is theoretical, in the sense that it assumes that an adversary has access to a MAC oracle over messages of arbitrary length. This assumption is unrealistic since neither the card nor the reader provide access to such a powerful oracle. Their second attack requires full control over a legitimate reader in order to issue arbitrary commands. Besides this assumption, it requires 2^{42} online authentication queries which, in practice, would take more than 710 years to gather. Our attacks, however, are practical in the sense that they can be executed within a day and require only wireless communication with a genuine iClass card/reader.

2.1. Research contribution

The contribution of this paper consists of several parts. First it describes the reverse engineering of the built-in key diversification algorithm of iClass Standard. The basic diversification algorithm, which also forms the basis for iClass Elite key diversification, 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.

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. To demonstrate this, we give the inverse function $hash0^{-1}$ that on input of a 64 bit bitstring outputs a modest amount (on average 4) of candidate pre-images. This results in our first attack on the iClass Standard key diversification that recovers a master key from an iClass reader which is of comparable complexity to that of breaking single DES. It only uses weaknesses in the key diversification algorithm. Since in the end it comes down to breaking DES, it can be accomplished within a few days on a RIVYERA (a generic massively parallel FPGA-computer, see http://www.sciengines.com). This is extremely sensitive since there is only one master key for all iClass Standard readers and from this master key all diversified card keys can be computed. As a faster alternative, it is possible to emulate a predefined card identity and use a DES rainbow table [Hel80], [Oec03] based on this identity to perform the attack. This allows an adversary to recover the master key even within minutes.

Furthermore, we have fully reverse engineered iClass's proprietary cipher and authentication protocol. This task of reverse engineering is not trivial since it was first necessary to bypass the read protection mechanisms of the microcontroller used in the readers in order to retrieve its firmware [GdKGVM12]. This process is explained later in Section 5. We also found serious vulnerabilities in the cipher that enable an adversary to recover the secret card key by just wirelessly communicating with the card. The potential impact of this second and improved attack against iClass Standard is vast since when it is combined with the vulnerabilities in the key diversification algorithm, which we exploited earlier, it allows an adversary to use this secret key to recover the master key. Additionally, we have reverse engineered the iClass Elite key diversification algorithm which we also describe in full detail. We show that this algorithm has even more serious vulnerabilities than the iClass Standard key diversification. In our third and last attack, an adversary is able to directly recover an "Elite" master key by simply communicating with a legitimate iClass reader.

Concretely, we propose three key recovery attacks: one on the iClass Standard key diversification, one against iClass Standard and one against iClass Elite. All attacks allow an adversary to recover the master key.

- The first attack, against iClass Standard key diversification, exploits the fact that the key diversification algorithm can be inverted. An adversary needs to let the genuine reader issue a key update command. The card key will be updated and from the eavesdropped communication the adversary learns the card key. The adversary proceeds by inverting the key diversification which results in a modest amount of pre-images. Now, only a bruteforce attack on single DES will reveal which master key was used.
- The second attack, against iClass Standard, exploits a total of *four* weaknesses in the cipher, key diversifi-

1. http://www.hidglobal.com/pr.php?id=393

cation algorithm and card implementation. In order to execute this attack the adversary first needs to eavesdrop one legitimate authentication session between the card and reader. Then it runs 2^{19} key updates and 2^{22} authentication attempts with the card. This takes less than six hours to accomplish (when using a Proxmark III as a reader) and recovers 24 bits of the card key. Finally, off-line, the adversary needs to search for the remaining 40 bits of the key. Having recovered the card key, the adversary gains full control over the card. Furthermore, computing the master key from the card key is as hard as breaking single DES and is done like in the first attack.

- The third attack, concerning iClass Elite, exploits *two* weaknesses in the key diversification algorithm and recovers the master key directly. In order to run this attack the adversary only needs to run 15 authentication attempts with a legitimate reader. Afterwards, off-line, the adversary needs to compute only 2^{25} DES encryptions in order to recover the master key. This attack, from beginning to end runs within 5 seconds on ordinary hardware.

We have executed all attacks in practice and verified these claims and attack times. These results, previously appeared in abbreviated form as [GdKGV11], [GdKGVM12].

2.2. Outline

This paper is organized as follows. Section 3 starts with a description of the iClass architecture, the functionality of the card, the cryptographic algorithms. Then, Section 4 describes the reverse engineering of the key diversification scheme that is used in iClass Standard. Here, we also give an attack against this iClass Standard key diversification that recovers the master key from a diversified key. This attack method forms the basis for the second attack against iClass Standard where it is used to recover the master key in its last step. The second attack itself is described in Section 6 after the reverse engineering and description of the cipher in Section 5. Finally, Section 7 describes the key diversification in iClass Elite and presents an attack against this scheme.

3. iClass

An HID iClass card is in fact a pre-configured and re-branded PicoPass card manufactured by Inside Secure[2]. HID configures and locks the cards so that the configuration settings can no longer be modified. This section describes in detail the functionality and security mechanisms of iClass and it also describes the reverse engineering process. Let us first introduce notation.

Notation: Throughout this article ε denotes the empty bitstring. $\oplus$ denotes the bitwise exclusive or. $\boxplus$ denotes addition modulo 256. $a \leftarrow b$ denotes the assignment of a value b to variable a. Given two bitstrings x and y, xy denotes their concatenation. Sometimes we write this concatenation explicitly with $x \cdot y$ to improve readability. $\overline{x}$ denotes the bitwise complement of x. 0^n denotes a bitstring of n zero-bits. Similarly, 1^n denotes a bitstring of n one-bits. Furthermore, given a bitstring $x \in (\mathbb{F}_2^k)^l$, we denote with $x_{[i]}$ the i-th element $y \in \mathbb{F}_2^k$ of x. We write y_i to denote the i-th bit of y. For example, given the bitstring $x = \texttt{0x010203} \in (\mathbb{F}_2^8)^3$ and $y \leftarrow x_{[2]}$ then $y = \texttt{0x03}$ and $y_6 = 1$.

Remark 3.1 (Byte representation): Throughout this paper, bytes are represented with their most significant bit on the left. However, the least significant bit is transmitted first over the air during wireless communication (compliant with ISO/IEC 15693). This is the same order in which the bits are input to the cryptographic functions. In other words, `0x0a0b0c` is transmitted and processed as input `0x50d030`.

3.1. Functionality

iClass cards come in two versions called 2KS and 16KS with respectively 256 and 4096 bytes of memory. The memory of the card is divided into blocks of eight bytes as shown in Figure 1. Memory blocks 0, 1, 2 and 5 are publicly readable. They contain the card identifier *id*, configuration bits, the card challenge c_C and issuer information. Block 3 and 4 contain two diversified cryptographic keys $k1$ and $k2$ which are derived from two different master keys $\mathscr{K}1$ and $\mathscr{K}2$. These master keys are referred to in the documentation as debit key and credit key. The card only stores the diversified keys $k1$ and $k2$. The remaining memory blocks are divided into two areas, which are represented by the host software as applications. The size of these applications is defined by the configuration block.

The first application of an iClass card is the *HID application* which stores the card identifier, PIN code, password and other information used in access control systems. Read and write access to the HID application requires a valid mutual authentication using the cipher to prove knowledge of $k1$. The master key of the HID application is a global key known to all iClass Standard compatible readers. The globally used key $\mathscr{K}1$ is kept secret by HID Global and is not shared with any customer or industrial partner. Recovery of this key undermines the security of all systems using iClass Standard. Two methods have been proposed [Mer10], [GdKGV11] to recover this key. To circumvent the obvious limitations of having only a global master key, iClass Elite uses a different key diversification algorithm that allows having custom master keys. The details regarding iClass Elite can be found in Section 7.1. The second global master key $\mathscr{K}2$ is used in both iClass Standard and Elite systems and it is available to any developer who signs a non-disclosure agreement with HID global. It is possible to extract this key

2. http://www.insidesecure.com/eng/Products/Secure-Solutions/PicoPass

Block	Content	Description
0	Card serial number	Identifier *id*
1	Configuration	
2	e-Purse	Card challenge c_C
3	Key for application 1	Diversified debit key $k1$
4	Key for application 2	Diversified credit key $k2$
5	Application issuer area	
6...18	Application 1	HID application
19...n	Application 2	User defined memory

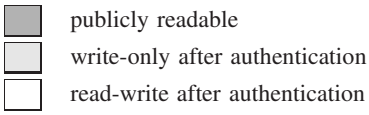


Figure 1: Memory layout of an iClass card

from publicly available software binaries [GdKGV11]. In addition, the document [HID06] contains this master key and is available online. This key $\mathscr{K}2$ can be used by developers to protect the second application, although in practice, $\mathscr{K}2$ is hardly ever used or modified.

The card provides basic memory operations like read and write. These operations have some non-standard behavior and therefore we describe them in detail.

- The `read` command takes as input an application number a and a memory block number n and returns the memory content of this block. This command has the side effect of selecting the corresponding key ($k1$ for application 1 or $k2$ for application 2) in the cipher and then it feeds the content of block n into the internal state of the cipher. Cryptographic keys are not readable. When the block number n corresponds to the address where a cryptographic key is stored, then `read` returns a bitstring of 64 ones.
- The `write` command takes as input a block number n, an eight-byte payload p and a MAC of the payload $\mathrm{MAC}(k, n \cdot p)$, where k is a diversified card key. When successful, it writes p in memory and it returns a copy of p for verification purposes. This command has the side effect of resetting the internal state of the cipher. In addition, when the block number n corresponds to the address where a cryptographic key k is stored, the payload is XOR-ed to the previous value instead of overwriting it, i.e., it assigns $k \leftarrow k \oplus p$.

Therefore, in order to update a key k to k', the reader must issue a `write` command with $k \oplus k'$ as payload. In this way the card will store $k \oplus k \oplus k' = k'$ as the new key. On the one hand, this particular key update procedure has the special feature that in case an adversary eavesdrops a key update he is unable to learn the newly assigned key, provided that he does not know k. On the other hand this introduces a new weakness which we describe in Section 6.2.

Before being able to execute `read` or `write` commands on the protected memory of a card, the reader needs to get access to the corresponding application by running a successful authentication protocol described in Section 3.2. Cryptographic keys $k1$ and $k2$ can be seen as part of application 1 and 2, respectively. This means that in order to modify a key e.g., $k1$, the reader first needs to run a successful authentication with $k1$.

3.2. Authentication protocol

This section describes the authentication protocol between an iClass card and reader. This protocol is depicted in Figure 3 and an example trace is shown in Figure 2. First, during the anti-collision protocol, the reader learns the identity of the card *id*. Then, the reader chooses an application and issues a `read` command on the card challenge c_C.

This c_C is called 'e-purse' in the iClass documentation [IC04] and it is a special memory block in the sense that it is intended to provide freshness. In the next step, the reader issues an `authenticate` command. This command sends to the card a reader nonce n_R and a MAC of the card challenge c_C concatenated with n_R. This MAC is computed using a diversified card key k. Finally, the card answers with a MAC of c_C, n_R followed by 32 zero bits. For more details over the MAC function see Section 5.2.
After a successful authentication on c_C the reader is granted read and write access within the selected application.

Remark 3.2: Since the card lacks a pseudo-random number generator, the reader should decrement c_C after a successful authentication in order to provide freshness for the next authentication, see Figure 2. This is not enforced by the card. Note that c_C is treated differently in the sense that when the tag stores c_C it swaps the first and last 32 bits (for reasons that are unknown to us). Therefore `0xFDFFFFFFFFFFFFFF` is stored by the tag as `0xFFFFFFFFFDFFFFFF` as shown in Figure 2.

4. iClass Standard

In this paper we first reverse engineer the iClass Standard key diversification. Then, we describe its weaknesses in Section 4.3. Finally, we describe the first attack against iClass Standard in Section 4.4.

Our first approach for reverse engineering is in line with that of [GdKGM+08], [LST+09], [GvRVWS10] and consists

Sender	Hex	Abstract
Reader	0C 00 73 33	Read identifier
Tag	47 47 6C 00 F7 FF 12 E0	Card serial number *id*
Reader	0C 01 FA 22	Read configuration
Tag	12 FF FF FF E9 1F FF 3C	iClass 16KS configuration
Reader	88 02	Read c_C and select $k1$
Tag	FE FF FF FF FF FF FF FF	Card challenge c_C
Reader	05 00 00 00 00 1D 49 C9 DA	Reader nonce $n_R = 0, \mathrm{MAC}(k1, c_C \cdot n_R)$
Tag	5A A2 AF 92	Response $\mathrm{MAC}(k1, c_C \cdot n_R \cdot 0^{32})$
Reader	87 02 FD FF FF FF FF FF FF FF	Write on block 02 followed by
	CF 3B D4 6A	$\mathrm{MAC}(k1, 02 \cdot c_C - 1)$
Tag	FF FF FF FF FD FF FF FF	Update successful

Figure 2: Authenticate and decrement card challenge c_C using $k1 = \texttt{0xE033CA419AEE43F9}$

of analyzing the update card key messages sent by an iClass compatible reader while we produce small modifications on the key, just after the DES operation and just before it is passed to the fortification function *hash0*. We used an Omnikey reader that supports iClass. Still, we first had to bypass the encryption layer of the Omnikey Secure Mode that is used in its USB communication in order to control the reader messages [GdKGV11]. 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 http://www.proxmark.org.

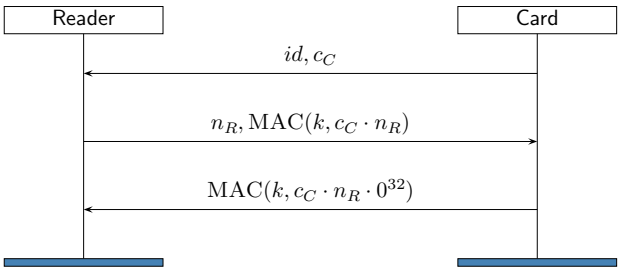


Figure 3: Authentication protocol

Later in Section 5, we use a different approach for reverse engineering the cipher and the key diversification for iClass Elite. In this approach we first recover the firmware from an iClass reader. Then, by disassembling the firmware we are able to recover the cipher and key diversification for iClass Elite. The knowledge about the structure of *hash0* which we describe in this section did help a lot in identifying the interesting parts of the firmware for reverse engineering.

4.1. Black box reverse engineering

This section describes how *hash0* [Cum06] (a.k.a. *h0* [Cum03]) was reverse engineered. The final description of *hash0* is given in Section 4.2. The method used to reverse engineer *hash0* studies the input-output relations of *hash0* in order to recover its internal structure. The primary goal is to learn how a card key k is derived from a master key $\mathscr{K}$ and the card identity *id*. The general structure of the key derivation is known. First, the iClass reader encrypts a card identity *id* with the master key $\mathscr{K}$, using single DES. The resulting ciphertext is then input to *hash0* which outputs the diversified key k.

$$k = \mathit{hash0}(\mathrm{DES}_{\mathrm{enc}}(\mathscr{K}, \mathit{id}))$$

We define the function *flip* that takes an input c and flips a specific bit in c. By flipping a bit we mean taking the complement of this bit. The definition *flip* is as follows.

Definition 4.1: Let the function *flip*: $\mathbb{F}_2^{64} \times \mathbb{N} \to \mathbb{F}_2^{64}$ be defined as

$$\mathit{flip}(c, m) = c_{63} \ldots c_{m+1} \cdot \overline{c_m} \cdot c_{m-1} \ldots c_0$$

Since we only learn the XOR difference between two *hash0* outputs we define the function *diff* that we use to express these XOR differences. The function *diff* computes the output difference of two *hash0* calls and is defined as follows.

Definition 4.2: Let the function *diff*: $\mathbb{F}_2^{64} \times \mathbb{N} \to \mathbb{F}_2^{64}$ be defined as

$$\mathit{diff}(c, m) = \mathit{hash0}(c) \oplus \mathit{hash0}(\mathit{flip}(c, m))$$

Now we use this definition of output difference to describe accumulative output differences of an input set $\mathscr{C}$.

$$k^{\vee m} = \bigvee_{c \in \mathscr{C}} \mathit{diff}(c, m), \qquad k^{\wedge m} = \bigwedge_{c \in \mathscr{C}} \mathit{diff}(c, m)$$

The results are grouped by the position of the flipped bit m. Then, the OR and AND is computed of all the results in a group. These cumulative OR and AND values for 64 bits that were flipped on a few thousand random 64-bit bitstrings $c \in \mathscr{C}$ are presented in Figure 5 and 6. The output difference for flipping all possible bits is abbreviated as follows.

$$k^{\vee} = \bigwedge_{m=0}^{63} k^{\vee m}, \qquad k^{\wedge} = \bigwedge_{m=0}^{63} k^{\wedge m}$$

4.1.1. Gathering input-output pairs. In this section we explain how we gather the input-output pairs for *hash0* and calculate the output differences. In our setup we have complete control over an iClass reader for which we can set and update the keys that are used. Furthermore, we are able to emulate iClass cards and learn all communication between the controlled reader and (emulated) iClass card. First, we analyze the input-output relations of *hash0* on bit level. This requires complete control over the input c of *hash0* which can be achieved in our test setup. In this test setup we emulate a card identity *id* and also know, or even can change, which master key $\mathscr{K}$ is used. 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 $c \in \mathbb{F}_2^{64}$.
- for each c
  - calculate $id = \mathrm{DES}_{\mathrm{dec}}(c, \mathscr{K})$ and $id^m = \mathrm{DES}_{\mathrm{dec}}(\mathit{flip}(c,m), \mathscr{K})$ for $m = 0 \ldots 63$
  - for each m authenticate with *id*, perform a key update, the reader requests the card identity again, now use id^m instead of *id*

Keep the master key $\mathscr{K}$ constant during the key updates described above. This delivers the XOR of two function evaluations of the form $\mathit{diff}(c,m) = \mathit{hash0}(c) \oplus \mathit{hash0}(\mathit{flip}(c,m))$. We performed this procedure for 3000 random values $c \in \mathscr{C}$. Experiments show that for this particular function, having more than 3000 samples does not produce any difference on the cumulative OR and AND tables. This amount of samples can be obtained within a couple of days.

4.1.2. Function input partitioning. Figure 5 shows the accumulated differences for the 48 rightmost output bits at input c. The results for the remaining 16 leftmost output bits are shown in Figure 6. These differences reveal that the input c of *hash0* is of the form $c = x \cdot y \cdot z_{[7]} \ldots z_{[0]}$ with $x, y \in \mathbb{F}_2^8$ and $z_{[i]} \in \mathbb{F}_2^6$. The eight output bytes are defined as $k_{[0]} \ldots k_{[7]}$ and constitute the diversified key k. We noticed that the first 16 bits of the input exhibit a different behavior that the rest and therefore decided to split the input in two parts. The structure of the mask in Figure 5 is computed with $x = y = 0^8$ and z a random bitstring. Whereas in Figure 6 we flip only bits of x and y. This leads to the following observations:

- $z_{[0]} \ldots z_{[3]}$ affects $k_{[0]} \ldots k_{[3]}$ and $z_{[4]} \ldots z_{[7]}$ affects $k_{[4]} \ldots k_{[7]}$.
- $z_{[0]} \ldots z_{[3]}$ and $z_{[4]} \ldots z_{[7]}$ generate a similar structure in the output but are mutually independent. This suggests the use of a subfunction that is called twice, once with input $z_{[0]} \ldots z_{[3]}$ and once with input $z_{[4]} \ldots z_{[7]}$. We call this function *check*.
- y_{7-i} affects $k_{[i]}$ for $i = 0 \ldots 7$. The OR-mask for y indicates a complement operation on the output while the AND-mask in Figure 6 shows that $k_{[i]_0}$ is exclusively affected by y for $i = 0 \ldots 7$.
- x defines a permutation. The output is scrambled after flipping a single bit within x. The AND-mask in Figure 6 shows that $k_{[i]_7}$ is exclusively affected by x for $i = 0 \ldots 7$.
- flipping bits in z never affects $k_{[i]_0}$ or $k_{[i]_7}$. This is inferred from the occurrences of `0x7e` (01111110 in binary representation) in Figure 5.

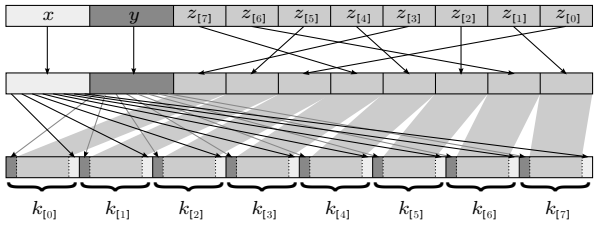


Figure 4: Schematic representation of the function *hash0*

	bit ↓	OR-mask of differences in output k	AND-mask of differences in output k
$z_{[0]}$	63	0x7e7e7e7e00000000	0x0400000000000000
	62	0x7e7e7e7e00000000	0x0400000000000000
	61	0x7a7e7e7e00000000	0x0800000000000000
	60	0x727e7e7e00000000	0x1000000000000000
	59	0x627e7e7e00000000	0x2000000000000000
	58	0x427e7e7e00000000	0x4000000000000000
	57	0x007e7e7e00000000	0x0000000000000000
$z_{[1]}$		...	...
	52	0x007e7e7e00000000	0x0000000000000000
$z_{[2]}$	51	0x00007e7e00000000	0x0000000000000000
		...	...
	46	0x00007e7e00000000	0x0000000000000000
$z_{[3]}$	45	0x0000007e00000000	0x0000000000000000
		...	...
	40	0x0000007e00000000	0x0000000000000000
$z_{[4]}$	39	0x00000000027e7e7e	0x0000000002000000
	38	0x00000000047e7e7e	0x0000000004000000
	37	0x00000000087e7e7e	0x0000000008000000
	36	0x00000000107e7e7e	0x0000000010000000
	35	0x00000000207e7e7e	0x0000000020000000
	34	0x00000000407e7e7e	0x0000000040000000
	33	0x00000000007e7e7e	0x0000000000000000
$z_{[5]}$		...	...
	28	0x00000000007e7e7e	0x0000000000000000
$z_{[6]}$	27	0x0000000000007e7e	0x0000000000000000
		...	...
	22	0x0000000000007e7e	0x0000000000000000
$z_{[7]}$	21	0x000000000000007e	0x0000000000000000
		...	...
	16	0x000000000000007e	0x0000000000000000

Figure 5: OR and AND-mask for flipping bits 16...63 of c

The above observations suggest that we can recover different parts of the function independently. Figure 4 summarizes how different parts of the input affect specific parts of the output. Note that from the last observation we know that the subfunction *check* operates on $z_{[i]_0} \dots z_{[i]_5}$ and affects $k_{[i]_1} \dots k_{[i]_6}$. Furthermore, it is observed that the leftmost bit of all output bytes $k_{[i]_7}$ and the permutation of $z_{[i]}$ to $k_{[i]_1} \dots k_{[i]_6}$ is determined by x. Finally, every input bit y_{7-i} is copied to output bit $k_{[i]_0}$.

Summarizing, *hash0* can be split into three different parts. The first part is the subfunction *check* which applies a similar operation on $z_{[0]} \dots z_{[3]}$ and $z_{[4]} \dots z_{[7]}$. In the second part a bitwise complement operation is computed based on bits from the input byte y. The last part applies a permutation that is defined by the input byte x. The following sections discuss the reverse engineering of these identified parts of *hash0*. Finally, the complete *hash0* definition is given in Section 7.

	bit ↓	OR-mask of differences in output k	AND-mask of differences in output k
y	15	0xfc00000000000000	0x8000000000000000
	14	0x00fc000000000000	0x0080000000000000
	13	0x0000fc0000000000	0x0000800000000000
	12	0x000000fc00000000	0x0000008000000000
	11	0x00000000fe000000	0x00000000fe000000
	10	0x0000000000fe0000	0x0000000000fe0000
	9	0x000000000000fe00	0x000000000000fe00
	8	0x00000000000000fe	0x00000000000000fe
x	7	0x7f7f7f7e7e7f7f7f	0x0101010000010101
	6	0x00007f7e7f000000	0x0000010001000000
	5	0x7f7e7e7e7f000000	0x0100000001000000
	4	0x7f7e7e7e7e7f0000	0x0100000000010000
	3	0x00007f7e7e7e7f00	0x0000010000000100
	2	0x7f7e7f7f7f7f7f00	0x0100010101010100
	1	0x7f7e7f7e7e7f7f00	0x0100010000010100
	0	0x7f7e7f7e7f7e7f00	0x0100010001000100

Figure 6: OR and AND-mask for flipping bits $0 \dots 15$ of c

4.1.3. Subfunction *check*. This section describes the reverse engineering of the subfunction *check* which operates on two times four 6-bit input values $z_{[0]} \dots z_{[3]}$ and $z_{[4]} \dots z_{[7]}$. In order to recover this part of the function we keep $x = y = 0^8$ and let z vary over random bitstrings. According to Figure 5 only flipping bits in z (positions 16 to 63 of input c) does matter for *check*. We write $modified(x)$ to indicate changes in x between two different test cases. We make modifications to the input and see where it affects the output. We start by looking at the following rules that are deduced from Figure 5.

$$\begin{aligned} modified(k_{[0]}) \rightarrow \quad & modified(z_{[7]}) \wedge \neg modified(z_{[0]} \dots z_{[6]}) \\ modified(k_{[4]}) \rightarrow \quad & modified(z_{[3]}) \wedge \neg modified(z_{[0]} \dots z_{[2]}) \\ & \wedge \neg modified(z_{[4]} \dots z_{[7]}) \end{aligned}$$

Note that $k_{[4]_1} \dots k_{[4]_6} = z_{[3]}$. For $k_{[0]}$ it is harder to find a function since flipping a single bit in $z_{[7]}$ may affect multiple bits in $k_{[0]}$. The relation between $z_{[7]}$ and $k_{[0]}$ becomes more clear when we look at specific input patterns and their corresponding output difference in Figure 7. The stars in the input pattern for $z_{[7]}$ denote a bit that can be either 0 or 1 without affecting the output difference of $k_{[0]}$. Note that, of course, the input bit that is being flipped can also be either 0 or 1 and is therefore also denoted by a star. We try to capture the output differences for flipping all possible bits between two different inputs c. We write z_7 when the bit flip is set to zero and $\check{z}_7$ when is set to one.

$z_{[7]}$ of c	$diff(c,63)_{[0]}$	$z_{[7]}$ of c	$diff(c,62)_{[0]}$
****0*	06	*****0	04
***01*	0e	***0*1	0c
**011*	1e	**01*1	1c
0111	3e	*011*1	3c
11111*	7c	0111*1	7c
01111*	7e	1111*1	7e

Figure 7: Input-output relations for $z_{[7]} \leftrightarrow k_{[0]}$

The difference $k^{\vee}_{[0]}$ based on flipping bits in $z_{[7]}$ is:

$$k^{\vee}_{[0]_1} \dots k^{\vee}_{[0]_6} = (z_7 \bmod 63) + 1 \oplus (\check{z}_7 \bmod 63) + 1$$

from which we deduce that

$$k_{[0]_1} \dots k_{[0]_6} = (z_7 \bmod 63) + 1 \tag{1}$$

The remaining $k_{[1]_1} \dots k_{[1]_6}$, $k_{[2]_1} \dots k_{[2]_6}$ and $k_{[3]_1} \dots k_{[3]_6}$ can be found in a similar way by flipping bits in the input and closely looking at the input-output relations. For more details on the reverse engineering of this function see [GdKGV11]. The complete definition of the function is given in Section 4.2. Eventually, the modulo operations are separated from the subfunction *check* and defined in the main function *hash0*. Also, the definition in Section 4.2 clarifies why the subfunction is called *check*. It checks equalities between the different components of z and affects the output accordingly.

4.1.4. Complement byte. The second byte of the input c is the complement byte y. It performs a complement operation on the output of the function as Figure 6 clearly shows. Flipping bit y_{7-i} results in the complement of $k_{[i]_0}$ in the output, for $i = 0 \dots 7$. Note that no other input bit influences any least significant output bit of the output bytes $k_{[i]_0}$. Furthermore, $k_{[i]_1} \dots k_{[i]_6}$ are flipped, however, keep in mind that we do not involve the action of byte x at this point, which prevents any permutation of the output.

Finally, every $k_{[i]_7}$ is not affected. It is important to observe that for $k_{[4]} \dots k_{[7]}$ the OR and AND-mask agree that the left 7 bits are always flipped while for $k_{[0]} \dots k_{[3]}$ this is not true. To be precise, the bits $k_{[i]_6}$ for $i = 0 \dots 3$ are *never* flipped. We found that the output value $z_{[j]}$ that constitutes output byte $k_{[i]}$ is decremented by one if $j \leq 3$ except when $y_{7-i} = 0$.

4.1.5. Permute byte. Finally, the byte x defines a permutation. Iterating over x while y and $z_{[0]} \dots z_{[7]}$ are constants shows that x is taken modulo 70. This follows from the fact

that the output values repeat every 70 inputs. The cumulative bitmasks of the output differences, shown in Figure 6, do not provide information about the permutation but do show that $k_{[i]_7}$ is affected. Experiments show that x is an injective mapping on $k_{[i]_7}$ for $i = 0 \dots 7$. This means that it is possible to learn x by looking at the least significant output bits $k_{[i]_7}$.

Furthermore, we conclude that the permutation is independent of y and z. This means that a permutation function *permute* can be constructed which takes x mod 70 as input and returns a particular mapping. We could recover this permutation because the values for $k_{[i]_7}$, for $i = 0 \dots 7$, directly relate to a unique mapping of the z input. The *hash0* function can be split up into *check* and *permute* subfunctions and is defined in Section 4.2.

4.2. The function *hash0*

The following sequence of definitions precisely describe the recovered function *hash0*. The details of this construction are not necessary to understand the attacks presented in Section 6.5 and Section 7.3.

The function *hash0* first computes $x' = x \bmod 70$ which results in 70 possible permutations. Then for all z_i the modulus and additions are computed before calling the subfunction *check*.

Then, the subfunction *check* 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 *check* is as follows.

Definition 4.3: Let the function $check\colon (\mathbb{F}_2^6)^8 \to (\mathbb{F}_2^6)^8$ be defined as

$$check(z_{[0]} \dots z_{[7]}) = ck(3,2,z_{[0]} \dots z_{[3]}) \cdot ck(3,2,z_{[4]} \dots z_{[7]})$$

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

$$ck(1,-1,z_{[0]} \dots z_{[3]}) = z_{[0]} \dots z_{[3]}$$

$$ck(i,-1,z_{[0]} \dots z_{[3]}) = ck(i-1,i-2,z_{[0]} \dots z_{[3]})$$

$$ck(i,j,z_{[0]} \dots z_{[3]}) = \begin{cases} ck(i,j-1,z_{[0]} \dots z_{[i]} \leftarrow j \dots z_{[3]}), & \textbf{if } z_{[i]} = z_{[j]}; \\ ck(i,j-1,z_{[0]} \dots z_{[3]}), & \text{otherwise.} \end{cases}$$

Definition 4.4: Define the function $permute\colon \mathbb{F}_2^n \times (\mathbb{F}_2^6)^8 \times \mathbb{N} \times \mathbb{N} \to (\mathbb{F}_2^6)^8$ as

$$permute(\varepsilon,z,l,r) = \varepsilon$$

$$permute(p_0 \dots p_n,z,l,r) = \begin{cases} (z_{[l]}+1) \cdot permute(p_0 \dots p_{n-1},z,l+1,r), & \textbf{if } p_n = 1; \\ z_{[r]} \cdot permute(p_0 \dots p_{n-1},z,l,r+1), & \text{otherwise.} \end{cases}$$

Definition 4.5: Define the bitstring $\pi \in (\mathbb{F}_2^8)^{35}$ in hexadecimal notation as

$$\pi = \texttt{0x0F171B1D1E272B2D2E333539363A3C474B}$$
$$\texttt{4D4E535556595A5C636566696A6C71727478}$$

Each byte in this sequence is a permutation of the bitstring `00001111`. Note that this list contains only the half of all possible permutations. The other half can be computed by taking the bit complement of each element in the list.
Finally, the definition of *hash0* is as follows.

Definition 4.6: Let the function $hash0\colon \mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8 \to (\mathbb{F}_2^8)^8$ be defined as $hash0(x,y,z_{[0]} \dots z_{[7]}) = k_{[0]} \dots k_{[7]}$ where

$$z'_{[i]} = (z_{[i]} \bmod (63-i)) + i \qquad i = 0 \dots 3$$

$$z'_{[i+4]} = (z_{[i+4]} \bmod (64-i)) + i \qquad i = 0 \dots 3$$

$$\hat{z} = check(z')$$

$$p = \begin{cases} \overline{\pi_{[x \bmod 35]}}, & \textbf{if } x_7 = 1; \\ \pi_{[x \bmod 35]}, & \text{otherwise.} \end{cases}$$

$$\tilde{z} = permute(p,\hat{z},0,4)$$

$$k_{[i]} = \begin{cases} y_{(7-i)} \cdot \overline{\tilde{z}_{[i]}} \cdot p_{(7-i)} + 1, & \textbf{if } y_{(7-i)} = 1; \\ y_{(7-i)} \cdot \tilde{z}_{[i]} \cdot \overline{p_{(7-i)}}, & \text{otherwise.} \end{cases} \qquad i = 0 \dots 7$$

This concludes the reverse engineering of the key diversification algorithm that is used in iClass Standard and defined as

$$k = hash0(\mathrm{DES}_{\mathrm{enc}}(\mathscr{K}, id)) \,.$$

4.3. Weaknesses in iClass Standard key diversification

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 4.4.

4.3.1. 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 little security. The shared key k_{CUW} that is used for this practice is the same for all Omnikey readers. This key is included in software that is publicly available online, which only gives a false feeling of security.

4.3.2. Weak key fortification. This section clarifies why *hash0* is not strengthening the diversified key k_{id} at all. First, note that only the modulo operations in *hash0* on x ($\frac{256}{70}$) and $z_{[0]}, \dots, z_{[7]}$ are responsible for collisions in the output. The expected number of pre-images for an output of *hash0* is given by

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

When we want to invert the function *hash0* we need to find the possible inputs that generate one specific output. Once we find a pre-image, we need to determine if there exist other

values within the input domain that lead 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. Furthermore, every permute byte x has at least two other values that map to the same output and in some cases there is even a third one. This means that the minimal number of pre-images is three. The probability q that for a given random input c there are only two other pre-images is

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

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

4.3.3. Inverting *hash0*. It is relatively easy to compute the inverse of the function *hash0*. Let us first compute the inverse of the function *check*. Observe that the function $check^{-1}$ is defined just as *check* except for one case where the condition and assignment are swapped, see Definition 4.7.

Definition 4.7: Let the function $check^{-1}\colon (\mathbb{F}_2^6)^8 \to (\mathbb{F}_2^6)^8$ be defined as $check(z_{[0]} \dots z_{[7]})$ in Definition 4.3 except for the following case where

$$ck^{-1}(i,j,z_{[0]} \dots z_{[3]}) = \begin{cases} ck^{-1}(i,j-1,z_{[0]} \dots z_{[i]} \leftarrow z_{[j]} \dots z_{[3]}), & \textbf{if } z_{[i]} = j; \\ ck^{-1}(i,j-1,z_{[0]} \dots z_{[3]}), & \text{otherwise.} \end{cases}$$

Definition 4.8: Define the function $permute^{-1}\colon \mathbb{F}_2^n \times (\mathbb{F}_2^6)^8 \times \mathbb{N} \times \mathbb{N} \to (\mathbb{F}_2^6)^8$ as

$$permute^{-1}(p,z,l=12,r) = \varepsilon$$

$$permute^{-1}(p,z,l<4,r) = \begin{cases} (z_{[r]}-1) \cdot permute^{-1}(p,z,l+1,r+1), & \textbf{if } p_r = 1; \\ permute^{-1}(p,z,l,r+1), & \text{otherwise.} \end{cases}$$

$$permute^{-1}(p,z,l\geq 4,r) = \begin{cases} z_{[l-4]} \cdot permute^{-1}(p,z,l+1,r), & \textbf{if } p_{l-4} = 0; \\ permute^{-1}(p,z,l+1,r), & \text{otherwise.} \end{cases}$$

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 4.9: Let the function $hash0^{-1}\colon (\mathbb{F}_2^8)^8 \to \{\mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8\}$ be defined as

$$hash0^{-1}(k_{[0]} \dots k_{[7]}) = \mathscr{C}$$

where

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

$$x' \text{ is unique elem. in } \mathbb{F}_2^8 \text{ s.t.} \begin{cases} p = \overline{\pi_{[x' \bmod 35]}} \Leftrightarrow x'_7 = 1 \\ p = \pi_{[x' \bmod 35]} \Leftrightarrow x'_7 = 0 \end{cases}$$

$$\begin{aligned} \dot{z}_{[i]} &= z'_{[i]} - (i \bmod 4) && i = 0 \dots 7 \\ z' &= check^{-1}(\hat{z}) \\ \hat{z} &= permute^{-1}(p, \tilde{z}, 0, 0) \\ \tilde{z}_{[i]} &= k'_{[i]_1} \dots k'_{[i]_6} && i = 0 \dots 7 \\ p_i &= \overline{k'_{[i]_7}} && i = 0 \dots 7 \\ k'_{[i]} &= \begin{cases} \overline{k_{[i]} - 1}, & \textbf{if } y_{(7-i)} = 1; \\ k_{[i]}, & \text{otherwise.} \end{cases} && i = 0 \dots 7 \\ y_i &= k_{[7-i]_0} && i = 0 \dots 7 \end{aligned}$$

4.3.4. Weak key diversification algorithm. The iClass Standard key diversification algorithm uses a combination of single DES and the proprietary function called *hash0*, which we reverse engineered. Based on our findings in the preceding sections, we conclude that the function *hash0* is not one-way nor collision resistant. In fact, it is possible to compute the inverse function $hash0^{-1}$ having a modest amount (on average 4) of candidate pre-images. After recovering a secret card key, recovering an iClass master key is not harder than a chosen plaintext attack on single DES. The use of single DES encryption for key diversification results in weak protection of the master key. This is a critical weakness, especially considering that there is only one master key for the HID application of all iClass cards. Furthermore, the composition of single DES with the function *hash0* does not strengthen the secret card key in any way. Even worse, when we look at the modulo operations that are applied on the z component of the *hash0* function input, we see that this reduces the entropy of the diversified card key with 2.23 bits.

4.4. Attacking iClass Standard key diversification

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 $\mathscr{K}$. In fact, the attack explained in this section requires one brute force run on DES. For this key recovery attack we need a strong adversary model where the adversary controls a genuine reader and is able to issue key update commands. Section 6.5 introduces an attack that allows a more restricted adversary. In this case, we use a strong adversary that

controls a genuine reader, like an Omnikey reader in Secure Mode. The adversary controls this reader and is able to issue key update commands. An attack consists of two phases and an adversary A needs to:

Phase 1.

- emulate a random identity id to the reader;
- issue an update key command that updates from a known user defined master key $\mathscr{K}'$ to the unknown master key $\mathscr{K}$ that A wants to recover. Now, A can obtain $k_{id} = hash0(\mathrm{DES}_{\mathrm{enc}}(\mathscr{K}, id))$ from the XOR difference;
- compute the set of pre-images $\mathscr{C}$ by $hash0^{-1}(k_{id})$;
- repeats Phase 1 until A obtains an output k_{id} with $|\mathscr{C}| = 3$.

Phase 2.

- A checks for every candidate DES key $\mathscr{K}^* \in \{0,1\}^{56}$ if $\mathrm{DES}_{\mathrm{enc}}(K^*, id) = c$, for every $c \in \mathscr{C}$;
- when the check above succeeds, A verifies the corresponding key $\mathscr{K}^*$ against another set of id and k_{id}.

We have verified this attack on the two master keys $\mathscr{K}1$ and $\mathscr{K}2$ that are stored in the Omnikey reader for the iClass application. The key $\mathscr{K}2$ 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 $\mathscr{K}1$ stored in software we were still able to verify it since we could dump the EEPROM of a reader where $\mathscr{K}1$ was stored, see Section 5.1. It would have been possible to recover *hash0* from the EEPROM as well, although the prior knowledge about *hash0* allowed us to identify more quickly where the remaining cryptographic functions were located in the EEPROM.

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 must be 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 total number of pre-images is 512.

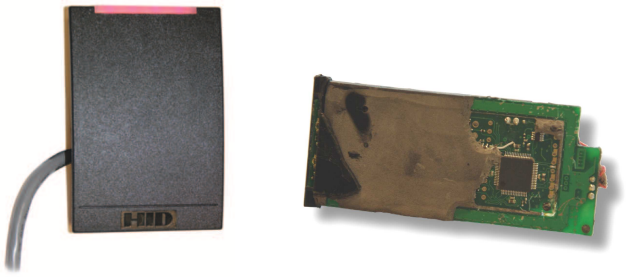
Finally, note that we need a strong adversary model in this attack. The adversary needs to control a genuine reader, by which we mean that the adversary is able to let the reader issue card key update commands. In a real-life setup this is not really feasible. The reverse engineering of the cipher and authentication protocol of iClass in Section 5 did not only reveal the iClass security mechanisms, but also more weaknesses that are described in Section 6. We use some of these weaknesses to lower the requirements on the adversary and deploy a second attack on iClass Standard, when the adversary does not control the reader, in Section 4.4.

5. The iClass cipher

This section first describes the reverse engineering process employed to recover the iClass cipher and to recover the iClass Elite key diversification algorithm. Then, we only describe the reverse engineered iClass cipher. We use this in Section 6 to mount a second (improved) attack on iClass Standard. The recovered key diversification for iClass Elite and its corresponding weaknesses lead to the third attack which is described in Section 7.

5.1. Firmware reverse engineering

In order to reverse engineer the cipher and the key diversification algorithm, we have first recovered the firmware from an iClass reader. For this we used a technique introduced in [Mer10] and later used in [GdKGV11]. Next, we will briefly describe this technique. iClass readers (Fig. 8), as many other embedded devices, rely on the popular PIC microcontroller (Fig. 8b) to perform their computations.



(a) iClass reader.

(b) iClass reader where the epoxy resin has been partially removed to expose the PIC microcontroller.

Figure 8: iClass readers

These microcontrollers are very versatile and can be flashed with a custom firmware. The (program) memory of the microcontroller is divided into a number of blocks, each of them having access control bits determining whether this block is readable/writable. Even when the PIC is configured to be non-writable, it is always possible to reset the access control bits by erasing the memory of the chip. At first glance this feature does not seem very helpful to our reverse engineering goals since it erases the data in the memory. Conveniently enough, even when the most common programming environments do not allow it, the microcontroller supports erasure of a single block. After patching the PIC programmer software to support this feature, it is possible to perform the following attack to recover the firmware:

- Buy two iClass RW400 (6121AKN0000) readers.
- Erase block 0 on one of the readers. This resets the access control bits on block 0 to readable, writable.

- Write a small dumper program on block 0 that reads blocks $1,\ldots,n$ and outputs the data via one of the microcontroller's output pins.
- Use the serial port of a computer to record the data. This procedure recovers blocks $1,\ldots,n$.
- Proceed similarly with the other reader, but erasing blocks $1,\ldots,n$. This in fact fills each block with NOP operations.
- At the end of block n write a dumper program for block 0.
- At some point the program will jump to an empty block and then reach the dumper program that outputs the missing block 0.

Once we have recovered the firmware, it is possible to use IDA Pro and MPLAB to disassemble, debug and reverse engineer the algorithms.

5.2. The cipher

This section describes the iClass cipher that we recovered from the firmware. This cipher is interesting from an academic and didactic perspective as it combines two important techniques in the design of stream ciphers from the '80s and beginning of the '90s, i.e., Fibonacci generators and Linear Feedback Shift Registers (LFSRs). The internal state of the iClass cipher consists of four registers as can be seen in Figure 9. Two of these registers, which we call left (l) and right (r) are part of the Fibonacci generator. The other two registers constitute linear feedback shift registers top (t) and bottom (b). In order to understand the description of the cipher correctly, take into account that the solid lines in Figure 9 represent byte operations while dotted lines represent bit operations.

Definition 5.1 (Cipher state): A *cipher state* of iClass s is an element of $\mathbb{F}_2^{40}$ consisting of the following four components:

- the *left register* $l = (l_0 \ldots l_7) \in \mathbb{F}_2^8$;
- the *right register* $r = (r_0 \ldots r_7) \in \mathbb{F}_2^8$;
- the *top register* $t = (t_0 \ldots t_{15}) \in \mathbb{F}_2^{16}$;
- the *bottom register* $b = (b_0 \ldots b_7) \in \mathbb{F}_2^8$.

The cipher has an input bit which is used (among others) during authentication to shift in the card challenge c_C and the reader nonce n_R. With every clock tick a cipher state s evolves to a successor state s'. Both LFSRs shift to the right and the Fibonacci generator iterates using one byte of the key (chosen by the $select(\cdot)$ function) and the bottom LFSR as input. During this iteration each of these components is updated, receiving additional input from the other components of the cipher. With each iteration, the cipher produces one output bit. The following sequence of definitions describes the cipher in detail; see also Figure 9.

Definition 5.2: The feedback function for the top register $T\colon \mathbb{F}_2^{16} \to \mathbb{F}_2$ is defined by

$$T(x_0 x_1 \ldots\ldots x_{15}) = x_0 \oplus x_1 \oplus x_5 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{14} \oplus x_{15}.$$

Definition 5.3: The feedback function for the bottom register $B\colon \mathbb{F}_2^8 \to \mathbb{F}_2$ is defined by

$$B(x_0 x_1 \ldots x_7) = x_1 \oplus x_2 \oplus x_3 \oplus x_7.$$

Definition 5.4 (Selection function):
The selection function $select\colon \mathbb{F}_2 \times \mathbb{F}_2 \times \mathbb{F}_2^8 \to \mathbb{F}_2^3$ is defined by

$$select(x,y,r) = z_0 z_1 z_2$$

where

$$\begin{aligned} z_0 &= (r_0 \wedge r_2) \oplus (r_1 \wedge \overline{r_3}) \oplus (r_2 \vee r_4) \\ z_1 &= (r_0 \vee r_2) \oplus (r_5 \vee r_7) \oplus r_1 \oplus r_6 \oplus x \oplus y \\ z_2 &= (r_3 \wedge \overline{r_5}) \oplus (r_4 \wedge r_6) \oplus r_7 \oplus x \end{aligned}$$

Definition 5.5 (Successor state): Let $s = \langle l,r,t,b \rangle$ be a cipher state, $k \in (\mathbb{F}_2^8)^8$ be a key and $y \in \mathbb{F}_2$ be an input bit. Define the *successor cipher state* $s' = \langle l',r',t',b' \rangle$ as

$$\begin{aligned} t' &\leftarrow (T(t) \oplus r_0 \oplus r_4) t_0 \ldots t_{14} \\ l' &\leftarrow (k_{[select(T(t),y,r)]} \oplus b') \boxplus l \boxplus r \\ b' &\leftarrow (B(b) \oplus r_7) b_0 \ldots b_6 \\ r' &\leftarrow (k_{[select(T(t),y,r)]} \oplus b') \boxplus l \end{aligned}$$

We define the successor function suc which takes a key $k \in (\mathbb{F}_2^8)^8$, a state s and an input $y \in \mathbb{F}_2$ and outputs the successor state s'. We overload the function suc to multiple bit input $x \in \mathbb{F}_2^n$ which we define as

$$\begin{aligned} \mathrm{suc}(k,s,\varepsilon) &= s \\ \mathrm{suc}(k,s,x_0 \ldots x_n) &= \mathrm{suc}(k,\mathrm{suc}(k,s,x_0 \ldots x_{n-1}),x_n) \end{aligned}$$

Definition 5.6 (Output): Define the function *output* which takes an internal state $s = \langle l,r,t,b \rangle$ and returns the bit r_5. We also define the function *output* on multiple input bits which takes a key k, a state s and an input $x \in \mathbb{F}_2^n$ as

$$\begin{aligned} output(k,s,\varepsilon) &= \varepsilon \\ output(k,s,x_0 \ldots x_n) &= output(s) \cdot output(k,s',x_1 \ldots x_n) \\ &\text{where } s' = \mathrm{suc}(k,s,x_0). \end{aligned}$$

Definition 5.7 (Initial state): Define the function *init* which takes as input a key $k \in (\mathbb{F}_2^8)^8$ and outputs the initial cipher state $s = \langle l,r,t,b \rangle$ where

$$\begin{aligned} t &\leftarrow \texttt{0xE012} & l &\leftarrow (k_{[0]} \oplus \texttt{0x4C}) \boxplus \texttt{0xEC} \\ b &\leftarrow \texttt{0x4C} & r &\leftarrow (k_{[0]} \oplus \texttt{0x4C}) \boxplus \texttt{0x21} \end{aligned}$$

*Definition 5.8 (*MAC *function):* Define the function $\mathrm{MAC}\colon (\mathbb{F}_2^8)^8 \times \mathbb{F}_2^n \to \mathbb{F}_2^{32}$ as

$$\mathrm{MAC}(k,m) = output(k,\mathrm{suc}(k,init(k),m),0^{32})$$

6. Weakness in iClass

This section describes weaknesses in the design and implementation of iClass. We present four weaknesses that are later exploited in Section 6.5 to mount a attack that recovers the systems master key.

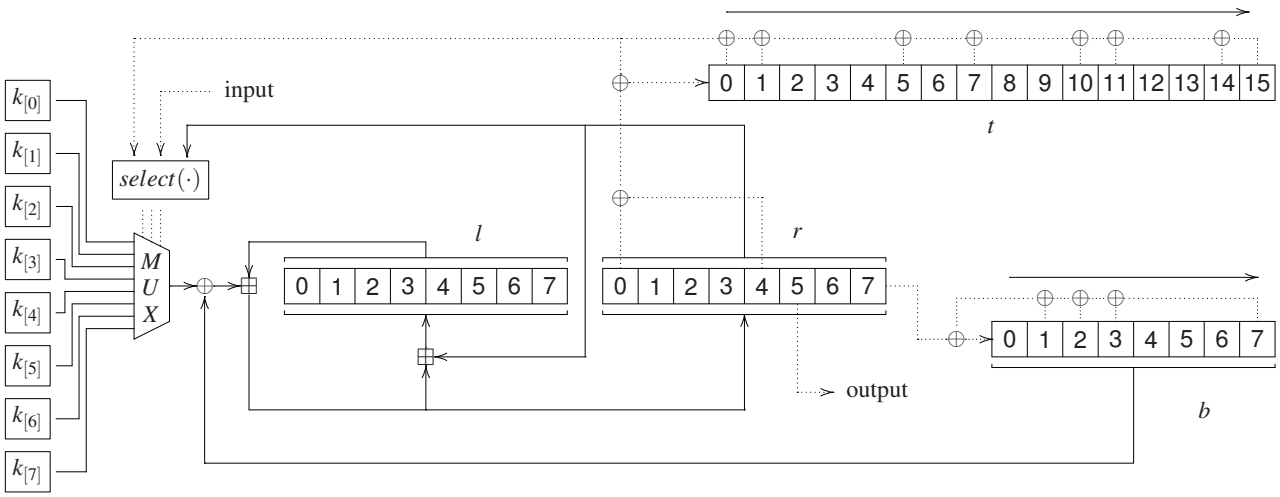


Figure 9: The iClass cipher

6.1. Weak keys

The cipher has a clear weakness when the three rightmost bits of each key byte are the same. Let us elaborate on that.

Proposition 6.1: Let β be a bitstring of length three. Then, for all keys $k \in \mathbb{F}_2^{64}$ of the form $k = \alpha_{[0]}\beta \ldots \alpha_{[7]}\beta$ with $\alpha_{[i]} \in \mathbb{F}_2^5$ the cipher outputs a constant C_β.

This is due to the fact that the output of the cipher is determined by the three rightmost (least significant) bits of register r, the three rightmost bits of l and the three rightmost bits of the selected key byte XOR b. Furthermore, only the rightmost bit of r influences register b. This means that the 5 leftmost bits of r and the 5 leftmost bits of each key byte affect only the key byte selection, but for the key under consideration this does not affect the output. The same holds for c_C and n_R as they are just input to the $select(\cdot)$ function. The following table shows the corresponding MAC value for each possible value of β.

The manufacturer seems to be aware of this feature of the cipher since the function *hash0*, used in key diversification, prevents such a key from being used. This weakness combined with the weakness described in Section 6.2 and 6.3 results in a vulnerability exploited in Section 6.5.

6.2. XOR key update weakness

In order to update a card key, the iClass reader does not send the new key to the card in the clear but instead it sends the XOR of the old and the new key (see Section 3.1). This simple mechanism prevents an adversary from eavesdropping the new key during key update. Although, this key update mechanism introduces a new weakness, namely, it makes it possible for and adversary to make partial modifications to the existing key. A key update should be an atomic operation. Otherwise it allows an adversary to split the search space in a time-memory trade-off. Moreover, in case the cipher has some weak keys like the ones described in Section 6.1, it allows an adversary to force the usage of one of these keys.

6.3. Privilege escalation

Several privilege escalation attacks have been described in the literature [KSRW04], [DDSW11]. The privilege escalation weakness in iClass concerns the management of access rights over an application within the card. After a successful authentication for application 1 has been executed, the reader is granted read and write access to this application. Then, it is possible to execute a `read` command for a block within application 2 without loosing the previously acquired access rights. More precisely, a `read` command on block n within application 2, with $n \neq c_C$, returns a sequence of 64 ones which indicates that permission is denied to read this block. Surprisingly, this read attempt on application 2 does not affect the previously acquired access rights on application 1. This `read` command though, has the side effect of loading the key $k2$ into the internal state of the cipher. In particular, from this moment on the card accepts `write` commands on application 1 that have a valid MAC computed using key $k2$.

6.4. Lower card key entropy

After careful inspection of the function *hash0* (Section 4.3.2) it becomes clear that this function attempts to fix the weak key weakness presented in this section.

The function *hash0* makes sure that, when looking at the last bit of each key byte, exactly four of them are zeros (and the other four of them are ones). Due to this restriction there

are only $\frac{8!}{(4!)^2} = 70$ possibilities for the last bits of each key byte, instead of $2^8 = 256$, reducing the entropy of the key by 1.87 bits. This constitutes the biggest part of the 2.23 bits entropy loss (Section 4.3.4) that is caused by *hash0*.

6.5. Key recovery attack on iClass Standard

This section shows how the weaknesses described in Section 6 can be exploited. Concretely, we propose an attack that allows an adversary to recover a card key by wirelessly communicating with a card and a reader. Once the card key has been recovered, the weak key diversification weakness described in Section 4.3 can be exploited in order to recover the master key. Next, we describe the attack in detail.

In order to recover a target card key $k1$ from application 1, an adversary A proceeds as follows. First, A eavesdrops a legitimate authentication trace on the e-purse with key $k1$, while making sure that the e-purse is not updated. If the reader attempts to update the e-purse, this can be prevented by playing as man-in-the-middle or by simply jamming the e-purse update message. Next, the adversary replays this authentication trace to the card. At this point the adversary gains read and write access to application 1. Although, in order to actually be able to write, the adversary still needs to send a valid MAC with $k1$ of the payload. To circumvent this problem, the adversary proceeds as described in Section 6.3, exploiting the privilege escalation weakness. At this point the adversary still has read and write access to application 1 but he is now able to issue `write` commands using MACs generated with the default key $k2$ [HID06] to write on application 1. In particular, A is now able to modify $k1$ at will. Exploiting the XOR key update weakness described in Section 6.2, the adversary modifies the card key $k1$ into a weak key by setting the three rightmost bits of each key byte the same. Concretely, the adversary runs $2^{3\times 7} = 2^{21}$ key updates on the card with $\Delta = 0^5\delta_{[0]}\dots 0^5\delta_{[6]}0^8 \in \mathbb{F}_2^{64}$ and $\delta_{[i]} = abc \in \mathbb{F}_2^3$ for all possible bits a, b and c. One of these key updates will produce a weak key, i.e., a key of the form $k = \alpha_{[0]}\beta\dots\alpha_{[7]}\beta$ with $\alpha_{[i]} \in \mathbb{F}_2^5$. Exploiting the weak key weakness described in Section 6.1, after each key update A runs 8 authentication attempts, one for each possible value of β, using the MAC values shown in Figure 10. Note that a failed authentication will not affect the previously acquired access rights. As soon as an authentication attempt succeeds, the card responds with a MAC value that univocally determines β as stated in Proposition 6.1. Knowing β, the adversary is able to recover the three rightmost bits of $k1_{[i]}$ by computing $\beta \oplus \delta_{[i]}$ for $i = 0\dots 6$. Furthermore, the three rightmost bits of $k_{[7]}$ are equal to $\beta \oplus 000 = \beta$. In this way, the adversary recovers $3\times 8 = 24$ bits of $k1$ and only has to search the remaining 40 bits of the key, using the legitimate trace eavesdropped in the beginning for verification.

This attack can be further optimized. The restriction on the last bit of each byte imposed by *hash0*, described at the end of Section 6.4, reduces the number of required key updates from 2^{21} to almost 2^{19}. Therefore, it reduces the total number of authentication attempts to $2^{19}\times 8 = 2^{22}$. Once the adversary has recovered the card key $k1$, as we already mention in Section 6.4, recovering the master key is just as hard as breaking single DES.

β	$C_\beta = \mathrm{MAC}(k, c_C \cdot n_R)$
000	`BF 5D 67 7F`
001	`10 ED 6F 11`
010	`53 35 42 0F`
011	`AB 47 4D A0`
100	`F6 CF 43 36`
101	`59 7F 4B 58`
110	`1A A7 66 46`
111	`E2 D5 69 E9`

Figure 10: Corresponding MAC for each value of β

7. iClass Elite

This section describes in detail the built-in key diversification algorithm of iClass Elite. Besides the obvious purpose of deriving a card key from a master key, this algorithm intends to circumvent weaknesses in the cipher by preventing the usage of certain 'weak' keys. In this way, it is patching a weakness in the iClass cipher. After the description of the iClass Elite key diversification in Section 7.1 we describe the weaknesses of this scheme in Section 7.2. Finally, the third and fastest attack of this paper, concerning iClass Elite, is given in Section 7.3.

First, recall the key diversification of the iClass Standard system that we described in Section 4.2. In this scheme, the iClass reader first encrypts the card identity *id* with the master key $\mathscr{K}$, using single DES. The resulting ciphertext is then input to a function called *hash0* which outputs the diversified key k, i.e., $k = hash0(\mathrm{DES}_{\mathrm{enc}}(\mathscr{K}, id))$. Here the DES encryption of *id* with master key $\mathscr{K}$ outputs a cryptogram c of 64 bits. These 64 bits are divided as $c = \langle x, y, z_{[0]}, \dots, z_{[7]}\rangle \in \mathbb{F}_2^8 \times \mathbb{F}_2^8 \times (\mathbb{F}_2^6)^8$ which is used as input to the *hash0* function. This function introduces some obfuscation by performing a number of permutations, complement and modulo operations. Besides that, it checks for and removes patterns like similar key bytes, which could produce a strong bias on the cipher. Finally, the output of *hash0* is the diversified card key $k = k_{[0]}, \dots, k_{[7]} \in (\mathbb{F}_2^8)^8$.

Remark 7.1: The DES implementation used in iClass is non-compliant with the NIST standard [FIP99]. Concretely, iClass deviates from the standard in the way of representing keys. According to the standard a DES key is of the form $\langle k_0 \dots k_6 p_0, \dots, k_{47} \dots k_{55} p_7\rangle$ where $k_0 \dots k_{55}$ are the actual key bits and $p_0 \dots p_7$ are parity bits. Instead, in iClass, a DES key is of the form $\langle k_0 \dots k_{55} p_0 \dots p_7\rangle$.

7.1. Key diversification on iClass Elite

The iClass Elite system is sold as a more secure and advanced solution than the iClass Standard variant. HID introduces iClass Elite (a.k.a. High Security) as the solution for "those who want a boost in security" [Cum03]. iClass Elite aims to solve the obvious limitations of having just one single world-wide master key for all iClass systems. Instead, iClass Elite allows customers to have a personalized master key for their own system. To this purpose, HID has modified the key diversification algorithm, described in Section 4.2 by adding an additional layer to it. This modification only affects the way in which readers compute the corresponding card key but does not change anything on the cards themselves. This section describes this key diversification algorithm in detail. Then, Section 7.2 describes two weaknesses that are later exploited in Section 7.3.

We first need to introduce a number of auxiliary functions and then we explain this algorithm in detail.

Definition 7.1 (Auxiliary functions): Let us define the following auxiliary functions. The bit-rotate left function

$$rl\colon \mathbb{F}_2^8 \to \mathbb{F}_2^8 \text{ as } rl(x_0 \dots x_7) = x_1 \dots x_7 x_0.$$

The bit-rotate right function

$$rr\colon \mathbb{F}_2^8 \to \mathbb{F}_2^8 \text{ as } rr(x_0 \dots x_7) = x_7 x_0 \dots x_6.$$

The nibble-swap function swap

$$swap\colon \mathbb{F}_2^8 \to \mathbb{F}_2^8 \text{ as } swap(x_0 \dots x_7) = x_4 \dots x_7 x_0 \dots x_3.$$

Definition 7.2: Let the function $hash1\colon (\mathbb{F}_2^8)^8 \to (\mathbb{F}_2^8)^8$ be defined as

$$hash1(id_{[0]} \dots id_{[7]}) = k_{[0]} \dots k_{[7]}$$

where

$$\begin{aligned}
k_{[i]} &= k'_{[i]} \bmod 128, && i = 0 \dots 7 \\
k'_{[0]} &= id_{[0]} \oplus \dots \oplus id_{[7]} && k'_{[4]} = \overline{rr(id_{[4]} \boxplus k'_{[2]})} + 1 \\
k'_{[1]} &= id_{[0]} \boxplus \dots \boxplus id_{[7]} && k'_{[5]} = \overline{rl(id_{[5]} \boxplus k'_{[3]})} + 1 \\
k'_{[2]} &= rr(swap(id_{[2]} \boxplus k'_{[1]})) && k'_{[6]} = rr(id_{[6]} \boxplus (k'_{[4]} \oplus \mathtt{3C})) \\
k'_{[3]} &= rl(swap(id_{[3]} \boxplus k'_{[0]})) && k'_{[7]} = rl(id_{[7]} \boxplus (k'_{[5]} \oplus \mathtt{C3}))
\end{aligned}$$

Definition 7.3: Define the rotate key function $rk\colon (\mathbb{F}_2^8)^8 \times \mathbb{N} \to (\mathbb{F}_2^8)^8$ as

$$\begin{aligned}
rk(x_{[0]} \dots x_{[7]}, 0) &= x_{[0]} \dots x_{[7]} \\
rk(x_{[0]} \dots x_{[7]}, n+1) &= rk(rl(x_{[0]}) \dots rl(x_{[7]}), n)
\end{aligned}$$

Definition 7.4: Let the function $hash2\colon (\mathbb{F}_2^8)^8 \to (\mathbb{F}_2^{64})^{16}$ be defined as $hash2(\mathscr{K}^{cus}) = y_{[0]} z_{[0]} \dots y_{[7]} z_{[7]}$ where

$$\begin{aligned}
z_{[0]} &= \mathrm{DES}_{enc}(\mathscr{K}^{cus}, \overline{\mathscr{K}^{cus}}) \\
z_{[i]} &= \mathrm{DES}_{dec}(rk(\mathscr{K}^{cus}, i), z_{[i-1]}) && i = 1 \dots 7 \\
y_{[0]} &= \mathrm{DES}_{dec}(z_{[0]}, \overline{\mathscr{K}^{cus}}) \\
y_{[i]} &= \mathrm{DES}_{enc}(rk(\mathscr{K}^{cus}, i), y_{[i-1]}) && i = 1 \dots 7
\end{aligned}$$

Next we introduce the Selected key. This key is used as input to the standard iClass key diversification algorithm. It is computed by taking a selection of bytes from $hash2(\mathscr{K}^{cus})$. This selection is determined by each byte of $hash1(id)$ seen as a byte offset within the bitstring $hash2(\mathscr{K}^{cus})$.

Definition 7.5: Let $h \in (\mathbb{F}_2^8)^{128}$. Let $k^{sel} \in (\mathbb{F}_2^8)^8$ be the Selected key defined as

$$h \leftarrow hash2(\mathscr{K}^{cus}); \qquad k^{sel}_{[i]} \leftarrow h_{[hash1(id)_{[i]}]} \qquad i = 0 \dots 7$$

The last step to compute the diversified card key is just like in iClass

$$k \leftarrow hash0(\mathrm{DES}_{enc}(k^{sel}, id))$$

7.2. Weaknesses in iClass Elite key diversification

This section describes two weaknesses in the key diversification algorithm of iClass Elite. These weaknesses are exploited in Section 7.3 to mount an attack against iClass Elite that recovers the custom master key.

7.2.1. Redundant key diversification on iClass Elite. Assume that an adversary somehow learns the first 16 bytes of $hash2(\mathscr{K}^{cus})$, i.e., $y_{[0]}$ and $z_{[0]}$. Then he can simply recover the master custom key $\mathscr{K}^{cus}$ by computing

$$\mathscr{K}^{cus} = \overline{\mathrm{DES}_{enc}(z_{[0]}, y_{[0]})}.$$

Furthermore, the adversary is able to verify that he has the correct $\mathscr{K}^{cus}$ by checking the following equality

$$z_{[0]} = \mathrm{DES}_{enc}(\mathscr{K}^{cus}, \overline{\mathscr{K}^{cus}}).$$

7.2.2. Weak key-byte selection on iClass Elite. Yet another weakness within the key diversification algorithm of iClass Elite has to do with the way in which bytes from $hash2(\mathscr{K}^{cus})$ are selected in order to construct the key k^{sel}. As described in Section 7.1, the selection of key bytes from $hash2(\mathscr{K}^{cus})$ is determined by $hash1(id)$. This means that only the card's identity decides which bytes of $hash2(\mathscr{K}^{cus})$ are used for k^{sel}. This constitutes a serious weakness since no secret is used in the selection of key bytes at all. Especially considering that, for some card identities, the same bytes of $hash2(\mathscr{K}^{cus})$ are chosen multiple times by $hash1(id)$. In particular, this implies that some card keys have significantly lower entropy than others. What is even more worrying, an adversary can compute by himself which card identities have this feature.

Card identity *id*	*hash1*(*id*)	Recovery
00 0B 0F FF F7 FF 12 e0	**01 01 00 00 45 01 45 45**	Byte 00, 01 in 2^{24}
00 04 0E 08 F7 FF 12 e0	**78 02** 00 00 45 01 45 45	Byte 02 in 2^{16}
00 09 0D 05 F7 FF 12 e0	**7B 03** 00 00 45 01 45 45	Byte 03 in 2^{16}
00 0A 0C 06 F7 FF 12 e0	**7A 04** 00 00 45 01 45 45	Byte 04 in 2^{16}
00 0F 0B 03 F7 FF 12 e0	**7D 05** 00 00 45 01 45 45	Byte 05 in 2^{16}
00 08 0A 0C F7 FF 12 e0	**74 06** 00 00 45 01 45 45	Byte 06 in 2^{16}
00 0D 09 09 F7 FF 12 e0	**77 07** 00 00 45 01 45 45	Byte 07 in 2^{16}
00 0E 08 0A F7 FF 12 e0	**76 08** 00 00 45 01 45 45	Byte 08 in 2^{16}
00 03 07 17 F7 FF 12 e0	**69 09** 00 00 45 01 45 45	Byte 09 in 2^{16}
00 3C 06 E0 F7 FF 12 e0	**20 0A** 00 00 45 01 45 45	Byte 0A in 2^{16}
00 01 05 1D F7 FF 12 e0	**63 0B** 00 00 45 01 45 45	Byte 0B in 2^{16}
00 02 04 1E F7 FF 12 e0	**62 0C** 00 00 45 01 45 45	Byte 0C in 2^{16}
00 07 03 1B F7 FF 12 e0	**65 0D** 00 00 45 01 45 45	Byte 0D in 2^{16}
00 00 02 24 F7 FF 12 e0	**5C 0E** 00 00 45 01 45 45	Byte 0E in 2^{16}
00 05 01 21 F7 FF 12 e0	**5F 0F** 00 00 45 01 45 45	Byte 0F in 2^{16}

Figure 11: Chosen card identities

7.3. Key recovery attack on iClass Elite

In order to recover a master key $\mathscr{K}^{cus}$, an adversary proceeds as follows. First, exploiting the weakness described in Section 7.2.2, the adversary builds a list of chosen card identities like the ones shown in Figure 11. This Figure contains a list of 15 card identities and their corresponding key-byte selection indices *hash1*(*id*). The selection of card identities in this list is malicious. They are chosen such that the resulting key k^{sel} has very low entropy (in fact, it is possible to find several lists with similar characteristics).

For the first card identity in the list, the resulting key k^{sel} is built out of only three different bytes from *hash2*($\mathscr{K}^{cus}$), namely 0x00, 0x01 and 0x45. Therefore, this key has as little as 24 bits of entropy (instead of 56). Next, the adversary will initiate an authentication protocol run with a legitimate reader, pretending to be a card with identity $id = 0x000B0FFFF7FF12E0$ as shown in the list. Following the authentication protocol, the reader will return a message containing a nonce n_R and a MAC using k. The adversary will repeat this procedure for each card identity in the list, storing a tuple $\langle id, n_C, n_R, \text{MAC} \rangle$ for each entry. Afterwards, off-line, the adversary tries all 2^{24} possibilities for bytes 0x00, 0x01 and 0x45 for the first key identity. For each try, he computes the resulting k and recomputes the authentication run until he finds a MAC equal to the one he got from the reader. Then he has recovered bytes 0x00, 0x01 and 0x45 from *hash2*($\mathscr{K}^{cus}$).

The adversary proceeds similarly for the remaining card identities from the list. Although, this time he already knows bytes 0x00, 0x01 and 0x45 and therefore only two bytes per identity need to be explored. This lowers the complexity to 2^{16} for each of the remaining entries in the list. The bytes that need to be explored at each step are highlighted with boldface in the list. At this point the adversary has recovered the first 16 bytes of *hash2*($\mathscr{K}^{cus}$). Finally, exploiting the weakness described in Section 7.2.1, the adversary is able to recover the custom master key $\mathscr{K}^{cus}$ with a total computational complexity of 2^{25} DES encryptions.

8. Conclusion

We have shown that the security of several building blocks of iClass is unsatisfactory. Again, obscurity does not provide extra security and there is always a risk that it can be circumvented. In fact, experience shows that instead of adding extra security it often covers up 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 [LG98]. 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.

Furthermore, we have found many vulnerabilities in the cryptography and implementation of iClass that result in two key recovery attacks. Our first attack requires one eavesdropped authentication trace with a genuine reader (which takes about 10ms). Next, the adversary needs 2^{22} authentication attempts with a card, which in practice takes approximately six hours. To conclude the attack, the adversary needs only 2^{40} off-line MAC computations to recover the card key. The whole attack can be executed within a day. For the attack against iClass Elite, an adversary only needs 15 authentication attempts with a genuine reader to recover the custom master key. The computational complexity of

this attack is negligible, i.e., 2^{25} DES encryptions. This attack can be executed from beginning to end in less than five seconds. We have successfully executed both attacks in practice and verified the claimed attack times.

This article reinforces the point that has been made many times: security by obscurity often covers up negligent designs. The built-in key diversification and especially the function *hash0* is advertised as a security feature but in fact it is a patch to circumvent weaknesses in the cipher. The cipher is a basic building block for any secure protocol. Experience shows that once a weakness in a cipher has been found, it is extremely difficult to patch it in a satisfactory manner. Using a well known and community reviewed cipher is a better alternative. The technique described in [RSH+12] could be considered as a palliating countermeasure for our first attack. More is not always better: the key diversification algorithm of iClass Elite requires fifteen DES operations more than iClass Standard while it achieves inferior security. Instead, it would have been more secure and efficient to use 3DES than computing 16 single DES operations in an ad hoc manner.

Furthermore, NIST have proposed a statistical test suite [RSN+01] that can be used to measure the cryptographic strength of a cipher. Although this might identify weaknesses in a cipher, still many weaknesses arise from mistakes in the implementation. In order to find these problems, it is good practice to incorporate some form of formal verification in the development and implementation of security products, see for instance [FL12]. Also, systematic and automated model checking techniques proposed in [Tre08] can help to detect and avoid implementation weaknesses like the privilege escalation in iClass. Alternatively, formalizing the whole design in a theorem prover [Bla01], [JWS11] may reveal additional weaknesses. It remains an open question whether the unusual data structures and functions that we recovered in this paper can be recovered using automated techniques, like for example with Howard [SSB11]. Automated techniques might speed up and assist in the reverse engineering of algorithms and data structures from software binaries. In line with the principles of responsible disclosure, we have notified the manufacturer HID Global and informed them of our findings back in November 2011. By the time of writing this article, HID has extended their product line with support for AES-enabled Mifare DESFire EV1 cards[3][4] which provide higher security levels for those customers considering migration alternatives. A practical counter-measure until migration would be to stop using the iClass Elite diversification scheme and only use iClass Standard with customized master keys for all applications. However, such measure should only be considered as a temporary mitigation and not as a definite solution as the (more expensive) attack on iClass Standard still applies.

3. www.hidglobal.com/iclass-hf-migration-reader-family-datasheet
4. www.hidglobal.com/mifare-desfire-ev1-card-datasheet

9. Acknowledgements

We are thankful to Milosch Meriac for his kind support while bypassing the PIC's read protection mechanisms which enabled the firmware recovery of the iClass reader. The authors would also like to thank the anonymous reviewers for their outstanding work. Their constructive and valuable comments helped us to substantially improve the quality of this paper.

References

[BGV+12] Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *12th Cryptographers' Track at the RSA Conference (CT-RSA 2012)*, volume 7178 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2012.

[BKZ11] Alex Biryukov, Ilya Kizhvatov, and Bin Zhang. Cryptanalysis of the Atmel cipher in SecureMemory, CryptoMemory and CryptoRF. In *9th Applied Cryptography and Network Security (ACNS 2011)*, volume 6715 of *Lecture Notes in Computer Science*, pages 91–109. Springer-Verlag, 2011.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE workshop on Computer Security Foundations (CSFW 2001)*, pages 82–96. IEEE Computer Society, 2001.

[Bog07] Andrey Bogdanov. Linear slide attacks on the KeeLoq block cipher. In *3rd International Conference on Information Security and Cryptology (INSCRYPT 2007)*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2007.

[COQ09] Nicolas T. Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the Hitag2 stream cipher. In *12th Information Security Conference (ISC 2009)*, volume 5735 of *Lecture Notes in Computer Science*, pages 167–176. Springer-Verlag, 2009.

[Cou09] Nicolas T. Courtois. The dark side of security by obscurity - and cloning MIFARE Classic rail and building passes, anywhere, anytime. In *4th International Conference on Security and Cryptography (SECRYPT 2009)*, pages 331–338. INSTICC Press, 2009.

[Cum03] Nathan Cummings. iClass levels of security, April 2003.

[Cum06] Nathan Cummings. Sales training. Slides from HID Technologies, March 2006.

[DDSW11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC 2010)*, volume 6531 of

Lecture Notes in Computer Science, pages 346–360. Springer-Verlag, 2011.

[DHW+12] Benedikt Driessen, Ralf Hund, Carsten Willems, Carsten Paar, and Thorsten Holz. Don't trust satellite phones: A security analysis of two satphone standards. In *33rd IEEE Symposium on Security and Privacy (S&P 2012)*, pages 128–142. IEEE Computer Society, 2012.

[dKGHG08] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the MIFARE Classic. In *8th Smart Card Research and Advanced Applications Conference (CARDIS 2008)*, volume 5189 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2008.

[FIP99] FIPS 46-3, Data Encryption Standard (DES). National Institute for Standards and Technology (NIST), Gaithersburg, MD, USA, 1999.

[FL12] Riccardo Focardi and Flaminia L. Luccio. Secure recharge of disposable RFID tickets. In *8th International Workshop on Formal Aspects of Security and Trust (FAST 2011)*, volume 7140 of *Lecture Notes in Computer Science*, pages 85–99. Springer-Verlag, 2012.

[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 *13th European Symposium on Research in Computer Security (ESORICS 2008)*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114. Springer-Verlag, 2008.

[GdKGV11] Flavio D. Garcia, Gerhard de Koning Gans, and Roel Verdult. Exposing iClass key diversification. In *5th USENIX Workshop on Offensive Technologies (USENIX WOOT 2011)*, pages 128–136. USENIX Association, 2011.

[GdKGVM12] Flavio D. Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, Lecture Notes in Computer Science. Springer-Verlag, 2012.

[Gol97] Jovan Dj. Golic. Cryptanalysis of alleged A5 stream cipher. In *16th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 1997)*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1997.

[GvRVWS09] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a MIFARE Classic card. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 3–15. IEEE Computer Society, 2009.

[GvRVWS10] 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 tradeoff. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[HID06] HID Global. HID management key letter, November 2006.

[HID09] HID Global. iClass RW100, RW150, RW300, RW400 readers, 2009.

[IC04] PicoPass 2KS. Product Datasheet, Nov 2004. Inside Contactless.

[ISO00] ISO/IEC 15693-1. Identification cards — Contactless integrated circuit cards — Vicinity cards — Part 1: Physical characteristics. International Organization for Standardization (ISO), Geneva, Switzerland, 2000.

[ISO06] ISO/IEC 15693-2. Identification cards — Contactless integrated circuit cards — Vicinity cards — Part 2: Air interface and initialization. International Organization for Standardization (ISO), Geneva, Switzerland, 2006.

[ISO09] ISO/IEC 15693-3. Identification cards — Contactless integrated circuit cards — Vicinity cards — Part 3: Anticollision and transmission protocol. International Organization for Standardization (ISO), Geneva, Switzerland, 2009.

[JWS11] Bart Jacobs and Ronny Wichers Schreur. Logical formalisation and analysis of the MIFARE Classic card in PVS. In *2nd International Conference on Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 2011.

[KJL+11] ChangKyun Kim, Eun-Gu Jung, Dong Hoon Lee, Chang-Ho Jung, and Daewan Han. Cryptanalysis of INCrypt32 in HID's iClass systems. Cryptology ePrint Archive, Report 2011/469, 2011.

[KKMP09] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking KeeLoq in a flash: on extracting keys at lightning speed. In *2nd International Conference on Cryptology in Africa, Progress in Cryptology (AFRICACRYPT 2009)*, volume 5580 of *Lecture Notes in Computer Science*, pages 403–420. Springer-Verlag, 2009.

[KSRW04] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *25th IEEE Symposium on Security and Privacy (S&P 2004)*, pages 27–40. IEEE Computer Society, 2004.

[LG98] Mike Loukides and John Gilmore, editors. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[LST+09] Stefan Lucks, Andreas Schuler, Erik Tews, Ralf-Philipp Weinmann, and Matthias Wenzel. Attacks on the DECT authentication mechanisms. In *9th Cryptographers' Track at the RSA Conference (CT-RSA 2009)*, volume 5473 of *Lecture Notes in Computer Science*, pages 48–65. Springer-Verlag, 2009.

[Mer10] Milosch Meriac. Heart of darkness - exploring the uncharted backwaters of HID iClass security. In *27th Chaos Computer Congress (27C3)*, December 2010.

[NESP08] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse engineering a cryptographic RFID tag. In *17th USENIX Security Symposium (USENIX Security 2008)*, pages 185–193. USENIX Association, 2008.

[Oec03] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. pages 617–630. Springer-Verlag, 2003.

[PN12] Henryk Plötz and Karsten Nohl. Peeling away layers of an RFID security system. In *16th International Conference on Financial Cryptography and Data Security (FC 2012)*, volume 7035 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, 2012.

[RSH+12] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *21st USENIX Security Symposium (USENIX Security 2012)*, pages 221–236. USENIX Association, 2012.

[RSN+01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. *NIST Special Publication (800-22)*, 22:1–152, 2001.

[SHXZ11] Siwei Sun, Lei Hu, Yonghong Xie, and Xiangyong Zeng. Cube cryptanalysis of Hitag2 stream cipher. In *10th International Conference on Cryptology and Network Security (CANS 2011)*, volume 7092 of *Lecture Notes in Computer Science*, pages 15–25. Springer-Verlag, 2011.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer-Verlag, 2009.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *18th Network and Distributed System Security Symposium (NDSS 2011)*, San Diego, CA, 2011. The Internet Society.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing (FORTEST 2008)*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, 2008.

[VGB12] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium (USENIX Security 2012)*, pages 237–252. USENIX Association, 2012.

[VGE13] Roel Verdult, Flavio D. Garcia, and Barış Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *22nd USENIX Security Symposium (USENIX Security 2013)*. USENIX Association, 2013.

[vN11] Petr Štembera and Martin Novotný. Breaking Hitag2 with reconfigurable hardware. In *14th Euromicro Conference on Digital System Design (DSD 2011)*, pages 558–563. IEEE Computer Society, 2011.