HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-2001-04

Protection of Keys against Modification Attack

Wai W. Fung
Department of Computer Science, HKUST
Clear Water Bay, Kowloon, Hong Kong
wwfung@cs.ust.hk

Mordecai J. Golin
Department of Computer Science, HKUST
Clear Water Bay, Kowloon, Hong Kong
golin@cs.ust.hk

James W. Gray, III
xaot88@aol.com

Abstract

In recent work, Anderson and Kuhn [3] described an attack against tamper-resistant devices wherein a secret key stored in EEPROM is compromised using a simple and low-cost attack. The attack consists of setting bits in the EEPROM using low-cost probes and observing the effect on the output of the device. These attacks are extremely general, as they apply to virtually any cryptosystem. The objective of the present work is to explore cryptographic techniques with the goal of raising the cost (in terms of time and money) of carrying out the EEPROM modification attack by Class I attackers, at least to a point where it is as prohibitive as the cost of purchasing more expensive equipment. We propose the m-permutation protection scheme in which the key will be encoded in a special way and burned into the EEPROM of the device. To attack the scheme, the attacker needs to be able to solve for $\boldsymbol{K}$ in the equation $\boldsymbol{K} = \bigoplus_{i=1}^{m} \boldsymbol{P}_i$ in which $\boldsymbol{P}_i$'s are unknown. It is observed that the m-permutation protection scheme does not distribute the key $\boldsymbol{K}$ uniformly. Analysis shows that $m = 3$ or $m = 5$ are already good enough practically to provide strong security if the encoding is done properly and that $m > 5$ may not give significant improvement to the security of the scheme.

1. Introduction

In 1997, Anderson and Kuhn introduced a low-cost attack called the EEPROM modification attack in [3]. This is a physical attack in which two low-cost microprobing needles [1] are used, with the help of a microscope and micromanipulators, to set or clear target bits in an effort to infer those bits. In addition to being low-cost, this attack is quite general and practical. The objective of the present work is to explore techniques of raising the cost (in terms of time and money) of carrying out an EEPROM modification attack, at least to the point where it is more expensive than EEPROM reading equipment.

In this attack, we assume that attackers can write arbitrary values to arbitrary locations of an EEPROM, and the location of the key within EEPROM is known. This is in fact often the case, since, in practice a DES key is often stored in the bottom eight bytes of the EEPROM. We also assume that EEPROM bits cannot be read directly since equipment to sense the value of an EEPROM bit is substantially more expensive than the microprobing needles.

Anderson and Kuhn's attack makes use of the key parity errors implemented in many applications utilizing DES. Their assumption is that the tamperproof device will not work (e.g., returning an error condition) whenever a key parity error is detected. We will see below that this assumption is not strictly necessary for a successful attack.

The paper is structured as follows. We first describe the EEPROM modification attack. Then various possible protection schemes will be discussed, and we argue that it is not an easy problem. Our proposed scheme for protecting the keys against the modification attack will be introduced in Section 5 and analysed in Section 6.

1.1. The EEPROM Modification Attack

Anderson and Kuhn's original attack proceeded as follows.

Modification Attack

```
loop for i from 0 to length(key) −1
    randomly set the i^th bit to 1 (or 0, it
    doesn't matter)
    operate (e.g. encrypt or sign) with the device
    if (device works) then
        conclude the bit was a 1
    if (key parity error message appears) then
        re-set the bit to 0 (as it was 0)
loopend
```

[1] A reviewer has pointed out that such microprobing needles can be obtained for only five US dollars each.

Although Anderson and Kuhn originally described the above attack with respect to a DES key and the associated key-parity bits, the attack can be generalized for an arbitrary key, with or without key-parity bits. In particular, to infer bit i, the attacker runs the device once before setting bit i, and once after setting bit i. If the output changes in any way (e.g., giving a key parity error or simply giving a different output) we know the original value for bit i is zero; if there was no change, the original value was one. Thus, the attack is quite general and efficient, and can be applied to virtually any key stored in a known EEPROM location.

2. A Closer Look

In our discussion, we use **K** to denote the actual key bit vector. That is, the key value to be used by the card in encrypting, signing, etc. **P** will be used to denote the physical key bit vector. It is the actual bit pattern stored in the EEPROM and is not necessarily the same as **K**. In particular, it may represent some encoding of **K** and may take up more space than **K** would.

There are at least two weaknesses that the modification attack exploits. The first is that $\mathbf{P} = \mathbf{K}$, that is, the key is stored bare in the device; and hence every bit compromised is the actual key bit to be used in the encryption process. The second is that the key parity error enables the attacker to know with *100% certainty* whether the current bit was changed or not. Together, these two weaknesses allow the attack to be performed fast, using $O(n)$ probes, where n is the length of **K**.

These observations lead us to consider possible protection schemes that could help close these loopholes. At first, it may seem that there are easy ways of foiling the modification attack. We will discuss a few ideas and why they do not work.

2.1. Hiding the Key in a Random Location

One may think it would help if we store the key in a random location; thus the attacker would not know where exactly, within the EEPROM, to apply his attack.

By the following reasoning, this approach adds negligible security to the system. Whenever the key needs to be used, its address (e.g., its offset within EEPROM) needs to be retrieved. That is, the actual address of the key needs to be stored on the card. But, is this address stored at some fixed location? If it is, the address becomes, essentially, part of the key; the attacker begins his attack by reading (via a modification attack) the address of the key and then continues by reading the actual key. If the address is not stored in a fixed location—perhaps it is also stored in a random location—then the address' address needs to be stored on the card. Now is the address' address stored in a fixed location?

Clearly, we cannot do address indirection ad infinitum; at some point, we need to store something in a fixed location. That something is, essentially, the key. Thus, storing the key in a random location, and using it indirectly, does not, in itself solve our problem. It succeeds in making the attacker's job a little bit harder because he needs to find the address before finding the actual key. But still, the attack can be done in $O(n)$ time. For the above reason, the model we set out in Section 3 assumes that the key is stored in a fixed location within EEPROM.

2.2. On-chip Reprogramming

Another approach that comes to mind immediately is for the card to keep track of the number of faults occurring during its use (using e.g., a counter) and erase the key once a certain threshold is reached. In fact, one can imagine any number of possible booby traps that could be set for the attacker, foiling with high probability any attempt to use an EEPROM modification attack.

This seems like a good solution. For example, if we erase the key the first time a key parity error is detected, the attacker would cause an error with probability $1 - 2^{-n}$ (for an n-bit key). Thus, the key would probably be erased by the third or fourth bit being attacked. For large n, the attacker's probability of obtaining the complete key would be negligible.

However, this approach again adds only a small amount of security. As pointed out by Anderson and Kuhn in [2], on-chip reprogramming of the EEPROM requires a programming voltage that would be generated using a large capacitor. Further, such capacitors can be identified under a microscope and destroyed, thus removing the on-chip EEPROM reprogramming capability of the card. Hence, the model we set out in the next section will rule out reprogramming of the EEPROM.

3. Model

We will make several assumptions in our discussion. Firstly, class I attackers (from the taxonomy of attackers proposed by IBM [1]), namely, clever outsiders with moderately sophisticated equipment, are assumed. In particular, we do not attempt to address attacks by insiders or attacks utilizing military-grade equipment. Secondly, it is assumed that **P** is stored in EEPROM and that the attacker cannot read the EEPROM directly. Finally, we assume the attacker is not able to see the exact wiring of the device. In particular, part of the wiring will be hidden beneath the surface of the chip (i.e., in one of the lower layers) during the chip fabrication process. This wiring is considered to be the "batch key", which is known only to the manufacturers and to those who need to program the device. For example, the devices would be manufactured in batches of 10,000 all with the same batch key. A single customer, say a bank, would purchase a batch of devices and would be given the batch key. This would enable them to program keys into the card.

On the other hand, we will assume that the attacker can get hold of the device and can operate it as many times as he wishes. Other than the hidden wiring, the algorithm is open and we assume the attacker knows the details of the protection scheme.

A protection scheme is formally specified by the following entities:

1. n — the length of the actual key **K**
2. p — the length of the physical key **P**
3. The function *encode* will be used at the card-programming/card-issuing organization (e.g., the bank); it will generate a pattern to be burned into the chip:

$$encode : \{0,1\}^n \longrightarrow \{0,1\}^p$$

4. The decoding functions and wiring functions will be implemented by the chip manufacturer. For each actual key bit, i, $0 \leq i < n$:
   - Define A_i to be the arity of the i^{th} decoding function. (Note that in practice, $A_i \geq 1$.)
   - The i^{th} decoding function $decode_i$ is the function producing the i^{th} bit of the actual key **K** given A_i bits of the physical key **P**.

   $$decode_i : \{0,1\}^{A_i} \longrightarrow \{0,1\}$$

   - The i^{th} wiring function determines the offset within **P** from where a wire is connected to the i^{th} decoding function:

   $$wiring_i : \{1, \cdots, A_i\} \longrightarrow \{0, 1, \cdots, p-1\}$$

   For example, $wiring_i(j) = k$ means the j^{th} input bit for the i^{th} decoding function is *wired* from the k^{th} bit of **P**.

For any *valid* protection scheme, we require that the same **K** will be decoded from its encoded version by the chip. That is, if the actual key is $\mathbf{K} = k_0 k_1 \cdots k_{n-1}$, and the physical key is $\mathbf{P} = encode(\mathbf{K})$, we require that for all i, $0 \leq i < n$,

$$k_i = decode_i(\mathbf{P}[wiring_i(1)], \cdots, \mathbf{P}[wiring_i(A_i)])$$

With respect to this model, the attacker is assumed to know the location of **P** as well as the decoding functions $decode_i$ and the wiring functions $wiring_i$. Of course, the card manufacturer can choose to keep all these secret. But protection schemes will be designed under Kerckoff's principle [6], in which the security of a cryptosystem is not based on the secrecy of what system is being used.

The attacker can use the microprobing needles to write a 0 or 1 to any location of the EEPROM storage for **P**. Each of the attacker's writes to the EEPROM is called a *probe*.

4. Possible Protection Schemes

4.1. Introducing Redundancy

In this approach, **P** is chosen to be a redundant representation of **K**. The idea is that even when some bits of **P** are changed, there will be no change in the output. It is tempting to think we will be able to design the wiring and decoding functions so that by the time an attacker is able to infer some bits of the actual key, other bits will be destroyed. In this way, the attacker would not be able to recover the entire key.

In this section, we illustrate this idea with an example that employs a voting function and conclude that for any deterministic decoding and wiring function, there would not be any real benefit. In particular, the attacker can always break the scheme in $O(p)$ time (where p is the length of the physical key).

4.1.1 Voting Scheme

A simple voting scheme can be set up as follows. We choose **P** to be three times the size of **K**. When a new device is to be programmed and issued with, say, an n-bit key, $\mathbf{K} = k_0 k_1 \cdots k_{n-1}$, we create **P** as follows. For each i, we will program three bits of **P** such that two of the bits are equal to k_i and the third is equal to the complement, $\overline{k_i}$. These three bits will be stored in locations $\mathbf{P}[3i]$, $\mathbf{P}[3i+1]$, and $\mathbf{P}[3i+2]$. However, the ordering of these three bits (i.e., which one will be the complement) will be chosen randomly at device programming time. This defines the $encode$ function in our model. In addition, this scheme also defines $p = 3n$, $\forall i, A_i = 3$, and $\forall i, j, wiring_i(j) = 3i + j - 1$.

To re-derive **K** from **P**, we can use the voting function which sets k_i to be the majority function of the values in the $(3i)^{th}, (3i+1)^{th}, (3i+2)^{th}$ locations of **P**. That is,

$$decode_i(P[3i], P[3i+1], P[3i+2]) =$$
bit value which occurs most often in the triple

Although somewhat more complicated than the previous approaches, the attacker can still successfully carry out an EEPROM modification attack on this scheme. The only difference is that the exact bit pattern of **P** might not be retrieved; instead, the bit pattern of a modified **P** will be recovered by the attacker. This is due to the fact that each decoding function, $decode_i$, is many-to-one. That is, $decode_i(1,1,0) = decode_i(1,0,1) = decode_i(0,1,1) = decode_i(1,1,1) = 1$; hence bit pattern 110 and any of its permutations, as well as 111 will all give the same result. Similarly, $decode_i(1,0,0) = decode_i(0,1,0) =$

$decode_i(0,0,1) = decode_i(0,0,0) = 0$. There, therefore, exists a number of possible bit patterns, $\mathbf{P}'$ that evaluate to each $\mathbf{K}$ under this scheme. However, this does not affect the attack. So long as the attacker obtains some $\mathbf{P}'$ that results in the actual value of $\mathbf{K}$, the attacker has succeeded. If during the EEPROM modification attack, the attacker ever modifies too many bits, a different $\mathbf{K}$ will be generated and a different output will be observed. At that point, the attacker can always correct the modified bit and go on. In this way, a usable $\mathbf{P}'$ (of length $3n$) can be found in only one pass of the $3n$ bits. Since (on average) half the key bits will need to be corrected, the expected number of probes in this attack is $(4.5)n$.

Some may think that the situation might improve if a more complicated function is designed and more redundancy is used in $\mathbf{P}$. But the above attack indicates that once there is a change in the output of the device, the attacker infers the value of a bit in $\mathbf{P}$. Thus, a modified $\mathbf{P}'$ can be found in $O(p)$ probes. This suggests the following proposition.

Definition 1 *A protection scheme is a* 5*-tuple* $(\mathcal{K}, \mathcal{P}, \mathcal{E}, \mathcal{W}, \mathcal{D})$ *in which*

1. $\mathcal{K}$ *is the* n*-bit actual key space* $\{0,1\}^n$ *and* $\mathbf{K} \in \mathcal{K}$ *is a* n*-bit key to be protected.*

2. $\mathcal{P}$ *is the* p*-bit physical key space* $\{0,1\}^p$ *and* $\mathbf{P} \in \mathcal{P}$ *is the bit stream stored in the EEPROM.*

3. $\mathcal{E}$ *is the set of encoding functions and* $\mathcal{E} : \mathcal{K} \longrightarrow \mathcal{P}$.

4. $\mathcal{W}$ *is the wiring function.*

5. $\mathcal{D}$ *is the decoding function.*

Proposition 1 *For any protection scheme* $(\mathcal{K}, \mathcal{P}, \mathcal{E}, \mathcal{W}, \mathcal{D})$*, if* $\mathcal{W}$ *and* $\mathcal{D}$ *are known, deterministic functions, then an attacker can break the protection scheme (that is, to find* $\mathbf{K}$*) in* $O(p)$ *probes.*

The proof, which is left out of this conference version, is by induction on p.

This proposition suggests that if we want a scheme that costs the attacker more than $O(p)$ probes, we must have something secret. Due to the Kerckoff's principle, we would not consider to use secret decoding functions. Rather, we will make use of hidden wirings to help the protection. We will implement a family of functions via the hidden wires so as to make the attacker time-expensive to find the particular function instance being used. One possible choice is the permutation.

4.2. Permutation

In this approach, the manufacturer chooses (as the batch key) a random permutation of the n-bit key. This permutation is used to form $\mathbf{P}$ at device programming time. This batch key is a very sensitive information, and the manufacturer must be working with the clients (e.g. banks) to make sure that this information can only be accessed by the fewest possible personels. Special management procedures must be designed to eliminate the possibilities of introducing class II attackers.

To restore the actual key $\mathbf{K}$, the wirings are used to invert the permutation. From the attacker's point of view, even though he does not know the permutation, he can proceed as follows. First, the attacker applies the original attack and, with n probings, finds the n bits of $\mathbf{P}$. At this point, the attacker does not know the permutation. Hence, he does not know the actual key, $\mathbf{K}$. However, if a *secret-key encryption scheme is used* in the device and the *encryption algorithm is known*, then the attacker can find the permutation in an additional $O(n)$ probes. In particular, the wiring pattern can be found as follows. As the attacker knows the function of the device (e.g., encryption using DES), he can find the device output corresponding to an arbitrarily selected input (using, e.g., a PC) using the following n (i.e., for DES $n = 56$) actual keys: $0\ldots01$, $0\ldots10$, $\cdots$, $10\ldots0$. Call these n outputs $\alpha_1, \ldots, \alpha_n$.

After computing the α_i, the attacker uses probes to write $0\ldots01$ to the area storing $\mathbf{P}$, operates the device, and compares the encrypted result with all the α_i. Since the protection scheme is simply a permutation, one of the α_i will match. Thus, the first wiring line is identified. Continuing with the remaining $n-1$ patterns ($0\ldots10$, $\cdots$, $10\ldots0$), all the wiring information can be revealed. Thus, the key $\mathbf{K}$ is found in $O(n)$ probes.

4.3. Protection via m Permutations

In all the protection attempts described above, the attacker can find the key in $O(n)$ probes (assuming p is a linear function of n). One may naturally wonder: is it possible to devise a scheme that can provide protection against the modification attack.

In this subsection, we show that by cascading (i.e., taking the cross product of) m permutations (for $m \geq 2$), we can achieve a design that will be more expensive for the attacker to perform the modification attack against.

Consider the case where $m = 2$. We proceed as follows.

- $p = 2n$.

- The device manufacturer chooses (randomly) two distinct permutation functions π_1 and π_2.

- Let $\mathbf{K}$ be the n-bit actual key. The chip will store $\mathbf{P} = \pi_1(\mathbf{K}) \cdot \pi_2(\mathbf{K})$ (where $\cdot$ denotes concatenation and in this paper, we overload the notation that $\pi_i(\mathbf{K}) = \text{permuted } \mathbf{K} \text{ with } \pi_i$).

- The wirings implement the the inverses of both permutations. In particular, $wiring_i(1) = \pi_1(i)$ and $wiring_i(2) = \pi_2(i) + n$.

- To restore the key, we require that for each i, **P**$[wiring_i(1)]$ = **P**$[wiring_i(2)]$. That is, if all n decoding functions receive matching inputs, a key is output; otherwise an error is given. In the case, where a key is output, $decode_i(x, x) = x$.

To restore the wirings w_a, an attacker can proceed as follows:

Attack

```
set all bits of P to 0
loop for i from 0 to n − 1
   set bit i of P to 1
   loop for j from 0 to n − 1
      set bit n + j of P to 1 and test
      if (ok) then
         record the wiring
         reset bit n + j of P to 0
         exit from this inner loop
      ifend
      reset bit n + j of P to 0
   loopend
   reset bit i of P to 0
loopend
```

On average, it will take $((n-1)+1)/2 = n/2$ trials to fix the 0^{th} wiring; $(n-1)/2$ trials for the first wiring and so on. This arithmetic sum is of the order of $n^2/4$. For example, with a 128-bit key, it is expected to take about 2^{12} probes to get the wiring information.

To further increase this number, we can cascade more permutations. From our investigations, the straightforward modification of the above attack will take the attacker $O(n^m)$ probes to find **K** with m permutations.

Proposition 2 *If a protection scheme uses m different permutations, cascaded as above, a brute-force search will take $O(n^m)$ time for the attacker to find **K**.*

The proof follows directly from the fact that $\sum i^{m-1}$ is of order $O(n^m)$.

5. Proposed Protection Scheme

5.1. Observation

In the previous section, we give an *upper bound* for breaking a whole batch of cards for the cascaded m-permutation protection scheme. Usually, we simply need to crack a single card instead of the whole batch of devices. With the above cascaded m-permutation scheme, it is hard to break the whole batch (i.e. to infer the wiring information), but this may not be true for the individual card where what we want to find is the key **K**.

One weakness of the cascaded m-permutation scheme is that the number of occurrences of 0's and 1's are preserved, though their locations are permuted. This gives the attacker additional information (the number of 0's and 1's in key **K**) to exploit. Before we proceed to discuss this, the following definitions are introduced:

Definition 2 *A permutation matrix[4] corresponding to a permutation π is a matrix M_π which has the effect of permuting a vector by π when it multiplies the vector. That is, $M_\pi \boldsymbol{K} =$ permuted **K**.*

M_π will be a $n \times n$ matrix if **K** is of length n, and **K** is considered as a $n \times 1$ column vector. The matrix $M_\pi = (m_{ij})_{n \times n}$ can be derived from π by

$$m_{ij} = \begin{cases} 1 & \text{if } j = \pi(i) \\ 0 & \text{otherwise} \end{cases}$$

Using this definition, we can describe the batch key for the cascaded m-permutation as a m-tuples $(M_{\pi_1}, M_{\pi_2}, \cdots, M_{\pi_m})$.

5.2. The scheme

To remove the weaknesses that we described in Section 5.1, we propose a scheme in which

1. the number of bit occurrence of 0's and 1's in **K** will be unknown to the attacker;

2. we do not care if the EEPROM can be read directly somehow.

The motivation for this scheme is that instead of storing a concatenation of the permuted versions of **K** as **P**, we store for each $\mathbf{P}_i$, a permuted version of **K** *xor*'ed with two independently chosen n-bit words $\widehat{\mathbf{K}}_i$ which will not be stored explicitly in the card. This will lead us to devise a scheme that satisfies the above two desired properties.

The basic set-up is the same as the cascaded m-permutation scheme but with the following amendments:

1. m must be odd and $m \geq 3$ (this will be needed later in (3))

2. **P** is no longer $M_{\pi_1}\mathbf{K} \cdot M_{\pi_2}\mathbf{K} \cdots M_{\pi_m}\mathbf{K}$; instead, it will be $\mathbf{P} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdots \mathbf{P}_m$ where

$$\mathbf{P}_1 \quad = \quad M_{\pi_1}(\mathbf{K} \oplus \widehat{\mathbf{K}}_2 \oplus \widehat{\mathbf{K}}_3)$$
$$\vdots$$

$$\begin{aligned}\mathbf{P}_i &= M_{\pi_i}(\mathbf{K} \oplus \widehat{\mathbf{K}}_{(i+1 \bmod m)} \oplus \widehat{\mathbf{K}}_{(i+2 \bmod m)}) \quad (1)\\ &\vdots\\ \mathbf{P}_m &= M_{\pi_m}(\mathbf{K} \oplus \widehat{\mathbf{K}}_1 \oplus \widehat{\mathbf{K}}_2)\end{aligned}$$

with $\widehat{\mathbf{K}}_0 = \widehat{\mathbf{K}}_m$.

In Section 5.3, we will show in property 3 that with this setup, there will be a total of 2^{n-1} possible bit patterns for **K** if the bit pattern of individual $\mathbf{P}_i$ does not contain too many 0's or 1's.

For each $\mathbf{P}_i$, **K** will be *xor*'ed with two different $\widehat{\mathbf{K}}_i$'s instead of one before it is permuted. This special arrangement enables us to easily decode **K**.

3. To decode **K** from the $\mathbf{P}_i$'s, we use the following:

Property 1 *If $\boldsymbol{P}_i$'s are set up as above, then*

$$\bigoplus_{i=1}^{m} M_{\pi_i}^{-1}\boldsymbol{P}_i = \begin{cases} \boldsymbol{K} & \text{if } m \text{ is odd} \\ 0 & \text{if } m \text{ is even}\end{cases} \quad (2)$$

As a result, if m is odd, the above decoding function will always return the correct value of **K** if the card has not been tampered with. The hidden wiring implements the M_{π_i}'s.

However, this is not an ideal decoding function as the attacker can compromise the i^{th} permutation details (i.e. find the M_{π_i}'s), via techniques described in Section 4.2, by comparing encrypted result of vector with only one bit on, i.e., by setting the $\mathbf{P}_i$ to $000\cdots1$, $000\cdots10$, $\cdots$, and $100\cdots0$ one by one while setting all other $\mathbf{P}_j$'s ($j \neq i$) to $000\cdots0$ and compare the result with the encrypted pattern of $000\cdots1$, $000\cdots10$, $\cdots$, and $100\cdots0$ respectively. Without loss of generality, suppose $\mathbf{P}_1 = 000\cdots1$ and $\mathbf{P}_i = 000\cdots0, \forall i > 1$. Then $\bigoplus_{i=1}^{m} M_{\pi_i}^{-1}\mathbf{P}_i = M_{\pi_1}^{-1}\mathbf{P}_1$ is a vector with only one bit on. As a result, the encrypted result using this key must be among the encrypted patterns of $000\cdots1, 000\cdots10, \cdots$, and $100\cdots0$ and we can therefore infer the wiring detail of this bit.

The basic problem with the above decoding function is that an attacker knows many valid physical keys (i.e., sets of $\mathbf{P}_i$'s) of a special form, and can use these to attack the card to find the wiring. Once the attacker knows the wiring, s/he can work backward to find the key. Instead, we need a method that is not expensive (in terms of complexity of building the circuit) to decode **K**, and for which it is difficult to find valid physical keys without knowledge of the physical wiring.

Our approach will be to introduce m new n-bit words $\widehat{\mathbf{P}}_1$, $\widehat{\mathbf{P}}_2, \cdots, \widehat{\mathbf{P}}_m$. These words will give us *another* equation for finding **K**. Comparing the result of this and equation (2) will enable the card to tell if it has been tampered with. These $\widehat{\mathbf{P}}_i$'s would be set to

$$\begin{aligned}\widehat{\mathbf{P}}_1 &= (M_{\pi_2}\mathbf{K}) \oplus \widehat{\mathbf{K}}_1\\ &\vdots\\ \widehat{\mathbf{P}}_i &= (M_{\pi_{(i+1 \bmod m)}}\mathbf{K}) \oplus \widehat{\mathbf{K}}_i \quad (3)\\ &\vdots\\ \widehat{\mathbf{P}}_m &= (M_{\pi_1}\mathbf{K}) \oplus \widehat{\mathbf{K}}_m\end{aligned}$$

where $\pi_0 = \pi_m$.

These $\widehat{\mathbf{P}}_i$'s will be stored in the EEPROM. Note that in this scheme the $\widehat{\mathbf{K}}_i$'s will be dumped after use. Their values can only be deduced when both **K** and the corresponding $\widehat{\mathbf{P}}_i$'s are known.

5.2.1 Decoding Function

The key **K** will then be computed as follows:

1. We calculate a first value for **K** via $\mathbf{K}_{init} = \bigoplus_{i=1}^{m} M_{\pi_i}^{-1}\mathbf{P}_i$
2. rearrange the m equations in equation (1) for each $\mathbf{P}_i$, multiply both sides of equation (1) by $M_{\pi_i}^{-1}$, leave **K** at the left hand side and then substitute in the appropriate values from equation (3) giving

$$\begin{aligned}\mathbf{K} &= M_{\pi_i}^{-1}\mathbf{P}_i \oplus \widehat{\mathbf{K}}_{(i+1 \bmod m)} \oplus \widehat{\mathbf{K}}_{(i+2 \bmod m)}\\ &= M_{\pi_i}^{-1}\mathbf{P}_i \oplus (\widehat{\mathbf{P}}_{(i+1 \bmod m)} \oplus M_{\pi_{(i+2 \bmod m)}}\mathbf{K}_{init})\\ &\qquad \oplus (\widehat{\mathbf{P}}_{(i+2 \bmod m)} \oplus M_{\pi_{(i+3 \bmod m)}}\mathbf{K}_{init})\end{aligned}$$

This yields m different expressions for **K**.

If $\mathbf{P}_i$'s have not been tampered with (that is, the card is functioning as it was first issued to the customer), the correct **K** will be returned if we logically *AND*, or logically *OR* all these m **K**'s. Hence, **K** can be calculated via the following steps:

1. Calculate $\mathbf{K}_{\text{and}}$ by:

$$\mathbf{K}_{\text{and}} = \bigwedge_{i=1}^{m}\Big\{ M_{\pi_i}^{-1}\mathbf{P}_i \oplus (\widehat{\mathbf{P}}_{(i+1 \bmod m)} \oplus M_{\pi_{(i+2 \bmod m)}}\mathbf{K}_{init}) \oplus (\widehat{\mathbf{P}}_{(i+2 \bmod m)} \oplus M_{\pi_{(i+3 \bmod m)}}\mathbf{K}_{init})\Big\} \quad (4)$$

2. Calculate $\mathbf{K}_{\text{or}}$ by:

$$\mathbf{K}_{\text{or}} = \bigvee_{i=1}^{m}\Big\{ M_{\pi_i}^{-1}\mathbf{P}_i \oplus$$

$$(\widehat{\mathbf{P}}_{(i+1 \bmod m)} \oplus M_{\pi_{(i+2 \bmod m)}}\mathbf{K}_{init}) \oplus$$
$$\left.(\widehat{\mathbf{P}}_{(i+2 \bmod m)} \oplus M_{\pi_{(i+3 \bmod m)}}\mathbf{K}_{init})\right\} (5)$$

3. If $\mathbf{K}_{\text{and}} = \mathbf{K}$ and $\mathbf{K}_{\text{or}} = \mathbf{K}$, then return $\mathbf{K}$; else return an error message.

With this decoding function, the attacker is unable to derive any information about the hidden permutation as before by by setting one $\mathbf{P}_i$ to $00\cdots1$ while setting the other $\mathbf{P}_j$'s ($j \neq i$) to $00\cdots0$ and $\widehat{\mathbf{P}}_i$ to $00\cdots0$. In this special case, $\mathbf{K} = 00\cdots1$. Define $\mathbf{1}_i$ to be the n-bit word with the i^{th} bit be 1 and all the other bits be 0. That is, $\mathbf{K} = \mathbf{1}_1$, while

$$\mathbf{K}_{\text{and}} = (\mathbf{1}_1 \oplus \mathbf{1}_{m_3} \oplus \mathbf{1}_{m_4}) \wedge (\mathbf{1}_{m_4} \oplus \mathbf{1}_{m_5}) \wedge \cdots \wedge (\mathbf{1}_{m_2} \oplus \mathbf{1}_{m_3})$$

and

$$\mathbf{K}_{\text{or}} = (\mathbf{1}_1 \oplus \mathbf{1}_{m_3} \oplus \mathbf{1}_{m_4}) \vee (\mathbf{1}_{m_4} \oplus \mathbf{1}_{m_5}) \vee \cdots \vee (\mathbf{1}_{m_2} \oplus \mathbf{1}_{m_3})$$

where $\mathbf{1}_{m_i} = M_{\pi_i}\mathbf{1}_1$.

To ensure that $\mathbf{K}_{\text{and}} = \mathbf{K}$, each sub-expression of $\mathbf{K}_{\text{and}}$ must have at least one 1. That is, $\mathbf{1}_{m_4} \neq \mathbf{1}_{m_5}, \cdots, \mathbf{1}_{m_2} \neq \mathbf{1}_{m_3}$. Consequently, $\mathbf{K}_{\text{or}}$ will have more than one 1 and hence $\mathbf{K}_{\text{or}} \neq \mathbf{K}$. As a result, the attacker cannot get any information by setting one $\mathbf{P}_i$ to $00\cdots1$ while setting the other $\mathbf{P}_j$'s ($j \neq i$) and $\widehat{\mathbf{P}}_i$'s to $00\cdots0$.

5.3. Some Properties

With the modification attack, the attacker can derive the values of the $\mathbf{P}_i$'s and $\widehat{\mathbf{P}}_i$'s but not the permutation details (that is, the π_i's are unknown). In this subsection, we are going to argue that these values of $\mathbf{P}_i$'s and $\widehat{\mathbf{P}}_i$'s cannot help much to derive the actual key $\mathbf{K}$. The attacker can perform an exhaustive search to find the value of $\mathbf{K}$ as follow (in this example, we assume that the card can do secret-key encryption):

1. The attacker picks a set of messages and then generates their corresponding ciphertexts with the working card.

2. Find the values of $\mathbf{P}_i$'s using the modification attack.

3. Then the attacker needs to find the value of $\mathbf{K}$. An attack is successful if given the pairs of message and ciphertext, the attacker can find the key $\mathbf{K}$ that encrypts the message to the corresponding ciphertext.

As it is known that the value of $\mathbf{K}$ is calculated based on $\mathbf{K} = \bigoplus_{i=1}^{m} M_{\pi_i}^{-1}\mathbf{P}_i$, we are going to show that in the worst case, there are a total of 2^{n-1} possible bit patterns for the correct n-bit key $\mathbf{K}$ and the attacker needs to test these 2^{n-1} possible bit patterns for $\mathbf{K}$. Although this process can be done off-line, it still takes a long time for the attacker to get $\mathbf{K}$.

We start with the following property on the binomial coefficients $C_i^n = \frac{n!}{i!(n-i)!}$ of $(1+x)^n$:

Property 2 $\sum_0^{\lfloor n/2 \rfloor} C_{2i}^n = \sum_0^{\lfloor (n+1)/2 \rfloor -1} C_{2i+1}^n = 2^{n-1}$

The following property asserts that equation (1) is safe. In particular, knowledge of $\mathbf{P}_i$'s alone cannot derive any information on $\mathbf{K}$:

Property 3 *If π_i's are unknown but fixed permutations, X and $\widehat{X}_i$'s are uniformly and independently chosen n-bit binary words, and Y satisfies $Y = M_{\pi_1}(X \oplus \widehat{X}_2 \oplus \widehat{X}_3)$, then $P(X = x|Y = y) = P(X = x)$.*

Property 4 *If π_1 and π_2 are two unknown but fixed permutations, $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ are two n-bit words (with n_1 and n_2 1's respectively), then $\boldsymbol{K}_{init} = M_{\pi_1}\boldsymbol{P}_1 \oplus M_{\pi_2}\boldsymbol{P}_2$ has a total of $\sum_{i=0}^{(u-l)/2} C_{l+2i}^n$ possible bit patterns where $l = |n_1 - n_2|$ and*

$$u = \begin{cases} n_1 + n_2 & \text{if } n_1 + n_2 \leq n \\ 2n - (n_1 + n_2) & \text{if } n_1 + n_2 > n \end{cases}$$

Furthermore, given $m \geq 3$, for any m n-bit words $\boldsymbol{P}_1, \boldsymbol{P}_2, \cdots, \boldsymbol{P}_m$ (with $n_1, n_2, \cdots$, and n_m 1's respectively), if there exist

1. *two words $\boldsymbol{P}_i$ and $\boldsymbol{P}_j$, $i \neq j$ such that the resulting $\boldsymbol{P}' = M_{\pi_i}\boldsymbol{P}_i \oplus M_{\pi_j}\boldsymbol{P}_j$ has possibly $l, l+2, \cdots$, and u 1's; and*

2. *a third word $\boldsymbol{P}_k$, $k \neq i, k \neq j$, such that $l \leq n_k \leq u$ and $n_i + n_j + n_k \geq n$,*

then $\boldsymbol{K}_{init} = \bigoplus_{i=1}^{m} M_{\pi_i}\boldsymbol{P}_i$ has a total of 2^{n-1} possible bit patterns.

The implication of this property is that there will be a total of 2^{n-1} possible bit patterns for the n-bit key $\mathbf{K}$ if the bit pattern of individual $\mathbf{P}_i$ does not contain too many 0's or 1's. Consequently, a brute force attack needs to try the 2^{n-1} possible bit patterns for $\mathbf{K}$ in the worst case if there are at least 3 $\mathbf{P}_i$'s satisfying the above properties, and the security is approximate to cracking for an unknown key of $n-1$ bit long. This does not imply, however, that the attacker has a probability of $1/2^{n-1}$ to guess for the key because the 2^{n-1} possible bit patterns for $\mathbf{K}$ are not uniformly distributed. In Section 6, we will discuss the probability distribution of $\mathbf{K}$, and analyze the consequences.

5.4. Further Improvement

The scheme described in Section 5.2 confuses the attackers by increasing the number of possible bit patterns to 2^{n-1}. A simple encoding algorithm can further increase this number to 2^n. The following algorithm encodes a given key $\mathbf{K}_{in}$ into the card:

Encode

input: $\mathbf{K}_{in}$

randomly choose m $\widehat{\mathbf{K}}_i$'s
if $\oplus_{i=1}^{m} \widehat{\mathbf{K}}_i$ has even number of 1's then
    $\mathbf{K} \longleftarrow \mathbf{K}_{in}$
else
    $\mathbf{K} \longleftarrow \overline{\mathbf{K}_{in}}$
endif
$\forall i, \mathbf{P}_i \longleftarrow$
    $M_{\pi_i}(\mathbf{K} \oplus \widehat{\mathbf{K}}_{(i+1 \bmod m)} \oplus \widehat{\mathbf{K}}_{(i+2 \bmod m)})$
$\forall i, \widehat{\mathbf{P}}_i \longleftarrow (M_{\pi_{(i+1 \bmod m)}} \mathbf{K}) \oplus \widehat{\mathbf{K}}_i$

return $\mathbf{P}_i$'s and $\widehat{\mathbf{P}}_i$'s

The idea exploits the fact that the attacker does not know the values of $\widehat{\mathbf{K}}_i$'s while we use this to determine if we encode the cryptographic key $\mathbf{K}_{in}$ or its complement $\overline{\mathbf{K}_{in}}$ into the card. As a result, the number of possible bit patterns is doubled. Hence the following proposition:

Proposition 3 *Under the same condition as in property 4, the above* **Encode** *procedure will increase the number of possible bit patterns for* $W = \bigoplus_{i=1}^{m} M_{\pi_i} W_i$ *to* 2^n.

6. Analysis

To attack the scheme described in Section 5.2, one has to guess the value of the key $\mathbf{K}$ (which is unknown) based on the values of the $\mathbf{P}_i$'s (which can be derived). Although the scheme can be designed to generate the whole spectrum of 2^n possible bit patterns, this does not mean that conditioned on the given values of the $\mathbf{P}_i$'s, each possible $\mathbf{K}$ is *uniformly* likely to have been the original key. Hence, we cannot draw the conclusion that the attacker only has a probability of $1/2^n$ of guessing the correct key when the attack is based only upon the knowledge of the $\mathbf{P}_i$'s.

We would study the chance that the attacker can guess the key. Using the modification attack, the attacker can see the values of the $\mathbf{P}_i$'s and knows that $\mathbf{K} = \bigoplus_{i=1}^{m} M_{\pi_i}^{-1} \mathbf{P}_i$. As the attacker cannot see the hidden wires, the values of $M_{\pi_i}^{-1}\mathbf{P}_i$'s are unknown. However, the permutation cannot change the number of 1's (and 0's) in the $\mathbf{P}_i$'s; this is the only additional information leaked to the attacker. Thus the only chance for the attacker to derive $\mathbf{K}$ is to take advantage of the properties of $\bigoplus_{i=1}^{m} M_{\pi_i}^{-1}\mathbf{P}_i$ (with the number of 1's in the $\mathbf{P}_i$'s known) and see if these properties can be exploited to derive something better than a brute force attack. As the $M_{\pi_i}^{-1}$'s are unknown to the attacker, the problem can be restated as follow:

> Let $n_1, n_2, \cdots, n_m$ be arbitrary integers where $0 \leq n_i \leq n$. Let $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_m$ be chosen independently with $\mathbf{P}_i$ being chosen uniformly from among the $C_{n_i}^{n}$ n-bit words with n_i 1's. Let $\mathbf{K} = \bigoplus_{i=1}^{m} \mathbf{P}_i$. What is the distribution of $\mathbf{K}$?

We will show that although the keys are not uniformly distributed, their distribution is close to uniform if weak values are discarded. Analysis also shows that $m = 3$ or $m = 5$ are already good enough practically to provide strong security if the encoding is done properly.

6.1. Approach

Suppose we are going to store an encoding of the n-bit key $\mathbf{K}$ into the device as described in Section 5.2, m n-bit $\mathbf{P}_i$'s with $n_1, n_2, \cdots, n_m$ 1's respectively will be created and burned into the EEPROM.

Notice that these m $\mathbf{P}_i$'s are generated randomly and independently. We can repeat the generation process until *all* these $\mathbf{P}_i$'s have some proven desired properties. That is, we can abandon the set of m $\mathbf{P}_i$'s if it is not generated satisfactorily (e.g. one of the $\mathbf{P}_i$'s contains a word with too many 0's or 1's), and repeat the process again (so that *all* $\mathbf{P}_i$'s satisfy the required condition that we will describe shortly).

If $\mathbf{P}_i$'s are chosen randomly, then n_i is a binomial random variable with parameters $n, p = 1/2$, and therefore the expected values of these n_i's will be $n/2$. We will study how n_i will deviate from $n/2$ and see if it can give us some hints on how to pick the $\mathbf{P}_i$'s.

6.2. Chernoff Bounds

Each $\mathbf{P}_i$ is made up of n bits. We can view these n bits in $\mathbf{P}_i$ as mutually independent Bernoulli variables, with n_i the sum of these 0's and 1's. According to the Chernoff bound [5], we have, for all δ, $0 < \delta < 1$,

$$Pr\left[\bigcup_{i=1}^{m}\left(\left|n_i - \frac{n}{2}\right| > \delta\frac{n}{2}\right)\right] < m \cdot 2\left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{n/2} = \gamma_{m,\delta}.$$

Chernoff bound arguments tell us that, with high probability, n_i is very close to $n/2$. This leads to the following idea. When generating $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_m$, we check if $\forall i, |n_i - n/2| < \delta n/2$ for some appropriately chosen δ (δ will be such that the probability that $\forall i, |n_i - n/2| > \delta n/2$ will be very high but small enough to substantially restrict the $\delta n/2$ size of the range that the n_i can appear in). If yes, we keep these P_i's and burn them into the device. If no, we throw away **all** of these $\mathbf{P}_i$'s and generate a new set of $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_m$. This process will continue until $|n_i - n/2| < \delta n/2$ for all i. For given m and δ, the expected number of times we need to generate $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_m$ is less than $1/(1 - \gamma_{m,\delta})$ and, if $\gamma_{m,\delta} < 1/2$, we will get a good set of $\mathbf{P}_i$'s in less than 2 trials on average.

6.3. Results

Given n-bit words $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_m$ with respectively $n_1, n_2, \cdots, n_m$ 1's, we would like to calculate the probability that key $\mathbf{K}$ has l 1's, which we will denote by $q(n, n_1, n_2, \cdots, n_m, l)$. For $m = 2$ case, we can exhaustively calculate the value of $q(n, n_1, n_2, l)$ using a computer program.

To derive the formulae for $q(n, n_1, n_2, \cdots, n_m, l)$ in terms of $q(n, n_1, n_2, l)$, we need the following basic lemma:

Lemma 1 *If B is a partition of X_i (that is, $B = \bigcup_i X_i$ and X_i's are mutually exclusive events), then for event A, $Pr(A|B) = \sum_i Pr(A|X_i)Pr(X_i|B)$.*

For the $m = 3$ case, we are given 3 $\mathbf{P}_i$'s with n_1, n_2, and n_3 1's respectively and we need to calculate the probability that the result $\mathbf{K}$ has l 1's:

$$
\begin{aligned}
& q(n, n_1, n_2, n_3, l) \\
\stackrel{\text{def}}{=} \; & Pr(|\mathbf{K}| = l | n_1, n_2, n_3) \\
= \; & \sum_{l_1} \Big\{ Pr(|\mathbf{K}| = l | (|\mathbf{P}_1 \oplus \mathbf{P}_2|) = l_1, n_3) \\
& \qquad Pr(|\mathbf{P}_1 \oplus \mathbf{P}_2| = l_1 | n_1, n_2, n_3) \Big\} \\
= \; & \sum_{l_1} q(n, l_1, n_3, l) q(n, n_1, n_2, l_1)
\end{aligned}
$$

where l_1 ranges from $|n_1 - n_2|$ to $\max(n_1 + n_2, n)$ step 2.

Using the argument from Subsection 6.2, we can choose the range for the n_i's and then calculate the probability for different l 1's in the resulting $\mathbf{K}$. Once we find $q(n, n_1, \cdots, n_m, l)$, then the probability for each individual key with l 1's is given by

$$
\alpha_{n, n_1, \cdots, n_m, l} = \frac{q(n, n_1, \cdots, n_m, l)}{C_l^n}. \tag{6}
$$

If $\alpha_{n, n_1, \cdots, n_m, l}$'s do not differ significantly, then the attacker will need to almost exhaustively search all the possible keys, as no one key will appear with a significantly higher probability.

A program has been written to calculate the values of $q(n, n_1, \cdots, n_m, l)/C_l^n$, for $m = 3$ and the result is summarised in Table 1. We showed in property 3 (of Section 5.3) that our protection scheme will give a total of 2^{n-1} possible bit patterns for a n-bit $\mathbf{K}$. If all these possible bit patterns are uniformly distributed, the attacker will only have a chance of $1/2^{127}$ of guessing the key $\mathbf{K}$ correctly. Table 1 shows that if the values of n_i's are restricted to a range from 32 to 96 (which corresponds to the Chernoff bound range given by taking $\delta = 0.5$), the weakest key will take the attacker $2^{112.1}$ trials to get it right on average, which, while not 2^{127}, is already good enough in practice. In addition, the number of *weak* keys (that is, keys with $\alpha > 1/2^{127}$) is less than that of the *strong* keys (that is, keys with $\alpha \leq 1/2^{127}$). As we further restrict the ranges for n_i and l, the results improve and α will be getting closer to $1/2^{n-1}$.

Similarly, for $m = 5$ case, we are given 5 $\mathbf{P}_i$'s with n_1, n_2, n_3, n_4 and n_5 1's respectively and we need to calculate the probability that the resulting $\mathbf{K}$ has l 1's and we have

$$
\begin{aligned}
& q(n, n_1, n_2, n_3, n_4, n_5, l) \\
\stackrel{\text{def}}{=} \; & Pr(|\mathbf{K}| = l | n_1, n_2, n_3, n_4, n_5) \\
= \; & \sum_{l_1} \Big\{ Pr(|\mathbf{K}| = l | (|\mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \mathbf{P}_3|) = l_1, n_4, n_5) \\
& \qquad Pr(|\mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \mathbf{P}_3| = l_1 | n_1, n_2, n_3) \Big\} \\
= \; & \sum_{l_1} q(n, l_1, n_4, n_5, l) q(n, n_1, n_2, n_3, l_1) \qquad (7) \\
= \; & \sum_{l_1} \Big\{ \sum_{l_2} q(n, l_2, n_5, l) q(n, l_1, n_4, l_2) \\
& \qquad \sum_{l_3} q(n, l_3, n_3, l_1) q(n, n_1, n_2, l_3) \Big\} \qquad (8)
\end{aligned}
$$

where l_1 takes the values of the possible number of 1's in the word $\mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \mathbf{P}_3$. Thus, we can calculate $q(n, n_1, n_2, n_3, n_4, n_5, l)$ in a number of ways. If we calculate it using equation (7), then we need huge memory (in $\Theta(n^4)$ space) to keep the table $q(n, \cdot, \cdot, \cdot, \cdot)$ values. If, instead, we use equation (8) to do the computation, it will take a very long time (in $\Theta(n^6)$ time) for the calculation. In

However, equation (7) has a special form that enables us to roughly estimate the performance of the scheme. As $\sum_{l_1} q(n, n_1, n_2, n_3, l_1) = 1$, we find that

$$
\begin{aligned}
& q(n, n_1, n_2, n_3, n_4, n_5, l) \\
= \; & \sum_{l_1} q(n, l_1, n_4, n_5, l) q(n, n_1, n_2, n_3, l_1) \\
\leq \; & \max_{l_1} q(n, l_1, n_4, n_5, l)
\end{aligned}
$$

Similarly,

$$
q(n, n_1, n_2, n_3, n_4, n_5, l) \geq \min_{l_1} q(n, l_1, n_4, n_5, l)
$$

Although we use n_4 and n_5 in the above two expressions, they do not necessarily correspond physically to the last two words stored in the EEPROM. Actually these bounds are true for any two n_i's among the m n_i's. (For example, it is also true that, $q(n, n_1, n_2, n_3, n_4, n_5, l) \geq \min_{l_1} q(n, l_1, n_1, n_4, l)$.)

As $q(n, l_1, n_4, n_5, l)$ will be close to $1/2^{n-1}$ when n_4 and n_5 are close to $n/2$, this implies that $q(n, n_1, n_2, n_3, n_4, n_5, l)$ will be close to $1/2^{n-1}$ if any two n_i's are close to $n/2$.

The same approach can be generalised for other values of m, and hence the following theorem:

Theorem 1 *For all $m \geq 3$, the probability that $|\boldsymbol{K}| = l$ given the n-bit words $\boldsymbol{P}_1, \boldsymbol{P}_2, \cdots, \boldsymbol{P}_{m+1}, \boldsymbol{P}_{m+2}$ having respectively $n_1, n_2, \cdots, n_{m+1}, n_{m+2}$ 1's is given by*

$$q(n, n_1, n_2, \cdots, n_m, n_{m+1}, n_{m+2}, l) = \sum_{l_1} q(n, l_1, n_{m+1}, n_{m+2}, l) q(n, n_1, \cdots, n_m, l_1)$$

where l_1 takes the values of the possible number of 1's in the word $\boldsymbol{P}_1 \oplus \cdots \oplus \boldsymbol{P}_m$. In addition, $q(n, n_1, n_2, \cdots, n_m, n_{m+1}, n_{m+2}, l)$ is bounded from above by

$$\max_{l_1} q(n, l_1, n_{m+1}, n_{m+2}, l).$$

To get an idea of how the bounds look, we restrict the values of n_{m+1} and n_{m+2} in the range from 45 to 83. In such case, for each (n_{m+1}, n_{m+2}) pair, we have

$$\max_{l_1} q(n, l_1, n_{m+1}, n_{m+2}, l) \leq \max_{n_{m+1}, n_{m+2} \in [45,83]; l_1} q(n, l_1, n_{m+1}, n_{m+2}, l)$$

We used a program to calculate all values of $q(128, l_1, n_{m+1}, n_{m+2}, l)$ where l_1 loops from 1 to 128; n_{m+1} and n_{m+2} loops from 45 to 83; and l loops from 1 to 128. For each l, we find the maximum and minimum values of $q(128, \cdot, \cdot, \cdot, l)$.

Definition 3 *The* bit equivalent *of a number v is defined as $-\log_2 v$.*

Fig. 1 shows the bit equivalent of the maximum and minimum values of $q(128, l_1, n_1, n_2, l)$ where n_1 and n_2 are in the range from 45 to 83. The upper curve represents the bit equivalent of the minimum value while the lower curve represents the bit equivalent of the maximum value over l. It shows that when an attacker is attempting to guess for the key **K** with l 1's, it is at least as difficult as guessing for a randomly chosen b-bits word, where b is bit equivalent of the maximum value for $q(128, \cdot, \cdot, \cdot, l)$. From the figure, it is observed that for a 128-bit key **K**, to guess for the weakest key is equivalent to guessing for a random key with more than 115-bit long. If we restrict the range to a narrower interval, say, from 50 to 78, the strength for the weakest key is equivalent to that of a random key with more than 121-bit long. That is, if any two $\mathbf{P}_i$'s fall into this range, we can be sure that the security of the scheme is equivalent to guessing for a random key with more than 121-bit long. This strength is reasonably good against class I attackers.

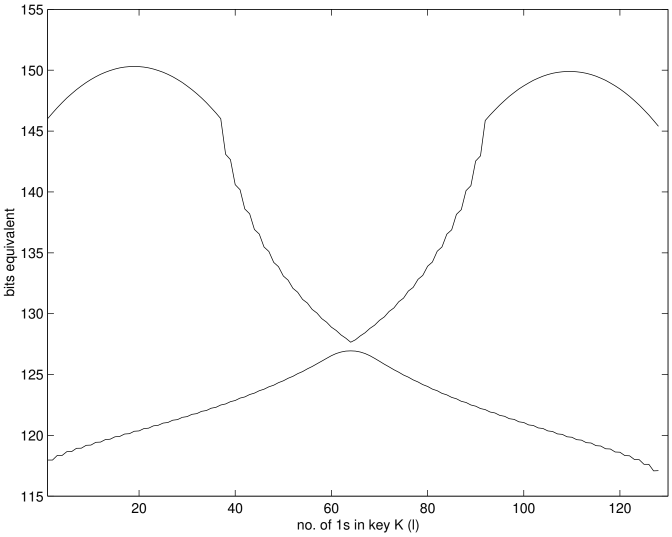

Figure 1. Bit equivalent of the max and min of $q(128, l_1, n_1, n_2, l)$ versus l (n_i's $\in [45, 83]$).

Theorem 1 shows that the overall security *may* not be improved by simply increasing m. Rather, we need to generate good $\mathbf{P}_i$'s. In practice, $m = 3$ or $m = 5$ are already good enough if the encoding is done properly; $m > 5$ may not give significant improvement to the security of the scheme — it only increases the chance of getting good $\mathbf{P}_i$'s.

7. Discussion and Conclusion

In this paper, we first introduced the modification attack and then studied and analysed ways of protecting keys stored in the EEPROM of tamper-resistant devices against the modification attack. We discussed some possible protection schemes such as voting schemes, and found that they are not very satisfactory. Instead, we introduced the idea of encoding the key rather than leaving it unencoded in the device. Our suggested scheme can be further enhanced to cover the whole spectrum of key values as discussed in Section 5.4.

In the device, m n-bit words $\mathbf{P}_i$'s are stored (together with another m n-bit auxiliary words for decoding); the actual key **K** will be decoded as $\bigoplus_{i=1}^{m} M_{\pi_i}^{-1} \mathbf{P}_i$ where M_{π_i} represents the permutation matrix for π_i implemented physically as hidden wires in the device. Keys in this scheme, however, are not uniformly distributed, leading to the possibility that the scheme can be *broken* by checking for high probability keys. We studied this problem and found that the distribution will be practically close to uniform, if all n_i's (the number of 1's in $\mathbf{P}_i$) are chosen properly (that is, relatively close to $n/2$ and furthermore, that it is easy to ensure that this condition holds). Theorem 1 asserts that for $m > 3$, the probability to break the scheme will be bounded above by $\max_{l_1} q(n, l_1, n_i, n_j, l)$ where $q(n, l_1, n_i, n_j, l)$ is the probability that the key $\mathbf{K}$ will have l 1 bits and $\mathbf{K}$ is the result of *xor*'ing three n-bit words with l_1, n_i and n_j 1 bits respectively. In particular, if n_i and n_j are in the range from 45 to 83, breaking the scheme is at least as difficult as breaking for a random key with more than 115-bit long. In addition, $m = 3$ or $m = 5$ are already good enough practically to provide strong security if the encoding is done properly and that $m > 5$ may not give significant improvement to the security of the scheme.

However, there remain a few areas of concern. First, if the attacker succeeds in breaking any one device, the whole batch to which this device belongs would become less secure. That is, the attacker could break any additional device belonging to the same batch in $O(n)$ time. This is simply because tamper-resistant devices will be mass produced, and the wiring topology will be the same for all devices in the same batch. Second, we should keep in mind that the scheme is insecure against class II attackers (i.e., knowledgeable insiders [1]) who may have knowledge of the wiring topology. Therefore, the device manufacturer must take appropriate procedures to protect this sensitive knowledge. Finally, against class III attackers (i.e., funded organizations [1]) there is really no/little hope of designing a truly tamper-resistant device, as they may possess equipment that makes it possible to completely reverse engineer the device.

m	range of n_i	range of l	$\max\alpha$	$\min\alpha$	number of $\alpha > 1/2^{127}$	number of $\alpha \leq 1/2^{127}$
3	32 – 96	1 – 128	$2^{-112.1}$	$2^{-155.4}$	2.26511e+43	7.07989e+43
		20 – 109	$2^{-115.8}$	$2^{-155.4}$	2.26511e+43	7.07989e+43
	38 – 90	1 – 128	$2^{-118.5}$	$2^{-152.3}$	1.25333e+43	3.81269e+43
		20 – 109	$2^{-120.7}$	$2^{-144.6}$	1.25333e+43	3.81269e+43
	45 – 83	1 – 128	$2^{-123.5}$	$2^{-136.5}$	5.03619e+42	1.5149e+43
		20 – 109	$2^{-124.3}$	$2^{-132.1}$	5.03619e+42	1.5149e+43
		45 – 83	$2^{-124.8}$	2^{-130}	5.03362e+42	1.51412e+43

Table 1. Maximum and minimum values of $\alpha_{n,n_1,n_2,n_3,l}$ for different ranges for l ($n = 128$).

Acknowledgements

The work of the first two authors was partially supported by HK RGC CERG grant HKUST 6137/98E

References

[1] D. Abraham, G. Dolan, G. Double, and J. Stevens. Transaction security system. *IBM Systems Journal*, 30(2):206–229, 1991.

[2] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11. USENIX Association, 1996.

[3] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *Security protocols : 5th International Workshop*, pages 125–136. Springer-Verlag, 1997.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[5] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1st edition, 1995.

[6] D. Stinson. *Cryptography : theory and practice*. CRC Press, 1995.