UICE: A High-Performance Cryptographic Module for SoC and RFID Applications

Ulrich Kaiser
Texas Instruments Deutschland GmbH
Haggertystr. 1
D 85350 Freising
email: d-kaiser@ti.com
URL: http://www.ti.com/ti-rfid/index.htm

Abstract

In order to overcome proprietary algorithms with respect to the system manufacturers, a free cryptographic module, the Universal Immobilizer Crypto Engine (UICE), will be proposed. This UICE algorithm is tailored to 8-bit microprocessor architectures and is therefore very fast in software and hardware. The dedicated hardware implementation leads to a small gate count, because the registers for input and output are shared. The important non-linear function, here an 8 x 8 S-Box, may be built as a gate array or small ROM with the advantage of flexibility. Several tests – statistical and random-number tests - have been performed in order to analyze the strength properties of the algorithm. So far no weakness was found after ten rounds of encryption.

Although this cryptographic module was intentionally developed for Radio-Frequency Identification (RFID) systems, it is a proper choice for all systems needing embedded cryptography such as SoC with bus encryption or firmware to be secured.

RFID-Systems have become commonplace in access control and security applications, the usage and importance of cryptographic co-processors in RFID transponder devices has grown significantly. Improved vehicle security systems, also known as immobilizers, are required due to increased vehicle theft worldwide. Such devices provide high security at low cost and low power.

Keywords: cryptographic co-processor, RFID, transponder, embedded system, car immobilizer, authentication, proprietary algorithm, non-proprietary algorithm, UICE, AES

1 Introduction

After illustrating the background behind the development of cryptographic modules for RFID-Systems, the current module DSG is shortly referenced. The next chapter describes the new module UICE, its design objectives, its properties and advantages, and the related C-code. Also comparisons with the well-known AES are given.

1.1 Background

Due to worldwide increases in automobile theft, there is a rapidly growing demand for less obtrusive, yet highly secure 'key-based immobilizers'. After the introduction of electronic immobilizers in 1993 theft numbers are decreasing again [12]. A second segment needing secure transactions, the Automatic Recognition of Customers, has been successfully introduced at gas stations (later at fast-food shops) in the U.S. [29].

Both systems are based on RFID technology [1, 13, 30]. They consist of a transponder and a reader (control unit, transceiver and antenna). The transponder is a tiny, battery-less, electronic device containning (at least) a small antenna and an IC that is of mixed-signal type and contains EEPROM, e.g. for the unique secret code, unique serial number, and configuration bits. Regarding frequency range the tags working at 13.56 MHz are flat and have mainly printed antennas; transponders at 125 or 134.2 kHz carry mainly ferrite or air coils with copper wires.

In the immobilizer application the driver places the ignition key in the ignition lock cylinder. The controller detects this and sends a command to the transceiver, which in turn sends out an RF energy burst; this activates the transponder, and then a command follows. After a few milliseconds, the transponder sends back its coded response (modulated RF telegram). The transceiver then demodulates this telegram and passes the code to the controller for verification. If the code is correct, the engine management computer is enabled.

The related system advantages are:
a) Security is based on a passive, contact-less device (no wear-out, no batteries),
b) unobtrusive to the user (no user actions required, transaction duration less than 200ms),
c) confined RF signals (no 'broadcast' to thieves (code grabbing)).

1.1.1 Immobilizer Operating Principle

The immobilizer system presented here uses a battery-less transponder working at 134.2 kHz (Fig.1). Data transmission is done by means of pulse width modulation (PWM) for the downlink (line of sight from transceiver to the transponder) and frequency-shift keying (FSK) for the uplink. The system works in half-duplex (HDX) mode [1]: First, energy and data are transmitted to and stored in the transponder. Second, the data-telegram is sent back to the listening reader unit.

Figure 1 RFID System Block Diagram

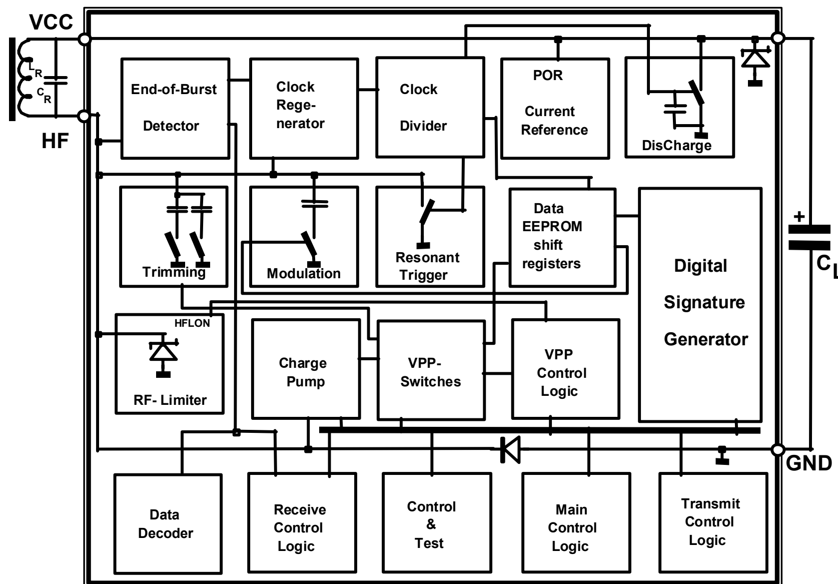


Figure 2 Block Diagram of the Integrated Circuit

The Immobilizer-IC block diagram is shown in Fig.2. The external LC tank L_R, C_R, serves to receive energy from the reader unit and for the data exchange. The external supply capacitor C_L stores energy, keeps it during burst pauses, especially for the uplink phase. The specialized low-power blocks for burst detection, clock recovery, trimming, maintenance of oscillation, charge pump, modulation, power-on-reset, discharge and limitation are described in [2, 3, 4] in detail.

1.2.2 Digital Signature Generator Transponder System

Various security levels are possible for immobilizers [5, 6, 12]:

a) **Fixed code transponders** are true read-only transponders with *unique* numbers, but with no reader-compatible, write-once version available.
b) **Rolling code transponders** modify the secret code in its EEPROM, but can suffer on synchronization problems.
c) **Password protected transponders** receive a unique code known only by the micro-controller of the vehicle.
d) **Digital Signature Generator transponders** (DSG) use a technique referred to as authentication, encryption, or challenge/response [5,6,11,12,16] to create the most advanced level of security for immobilizers.

The challenge/response operating principle is based upon the controller sending a random number (of pre-determined length) to the transponder, which is then scrambled in a unique way such that the controller can authenticate the response as being only from a valid transponder. Each transponder would scramble the challenge in a unique way, based on a hidden encryption key known only by the controller and transponder (Figure 3).

There are many advantages to this configuration [6] **:**
No portion of the critical encryption key code is ever transmitted (except once at installation and initialization in a secure environment). The encryption key code cannot be read-out or easily determined from the key alone, or a combination of the key and vehicle. Only the immobilizer controller knows what the next proper response will be. Transaction times are fast, i.e. below 150 ms.

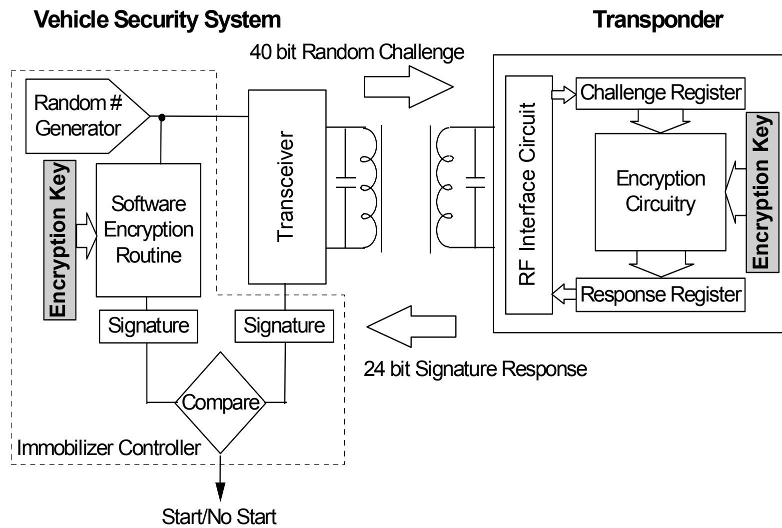


Figure 3. Digital Signature Generator Transponder System

Microprocessor software is simplified because no re-synchronization or password routines are needed. This device provides a 24-bit encrypted telegram (signature) in response to a 40 bit random number challenge. Intentionally, 16 bits of the 40-bit calculated result remain hidden in order to strengthen the encryption system against dictionary attacks [6,7,16].

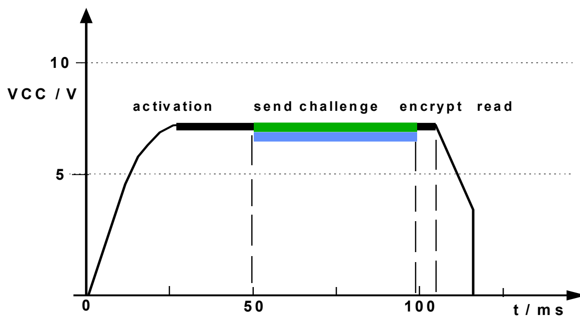


Figure 4. Transponder Charge Voltage Pattern

Figure 4 shows the characteristic supply voltage variation of the DSG during the authentication process. After charge-up of the supply capacitor, a command and the random challenge is written [3] into the transponder, and the Digital Signature Generator (DSG) is initialized. The reader unit provides further energy so, that the response (signature) is calculated after four hundred clock cycles (ca. 3ms). It is sent back as documented in [4] in about 15ms, and at the end the supply capacitor is discharged and the chip is reset internally in order to ensure that the device is accessible again quickly.

1.2 DSG Algorithm Design Objectives

All cryptographic algorithms can be broken. Security requirements are satisfied if the amount of effort to break the algorithm is uneconomic in relation to the possible gains.

The designers of the DSG algorithm assume that an attacker has a knowledge of cryptanalytic techniques, has a detailed knowledge of the algorithm, has access for several hours to the targeted transponder prior to an attempted violation, may use a computer over a long period at a private location to help break the system, and is not restricted to experiments with the transponder at the scene of the attempted violation.

The broad categories of attack considered are **Scanning, Dictionary Attack, and Cryptanalysis** [6-8, 16]. The Cryptanalysis methods considered include e.g.: Exhaustive Key Search, Key Homing, Correlation Attacks, Linearity Attacks, Linear Cryptanalysis, and Differential Cryptanalysis [6-8, 16]. The DSG algorithm is designed to frustrate these methods [10].

In order to fulfill the requirements of high speed, low-power, small area and compatibility to the HDX principle, a proprietary cryptographic algorithm was designed; this algorithm passes all known statistical tests successfully [10]. – This design developed in 1995 is the first cryptographic primitive embedded into an RFID device.

Hardware, such as an RSA processor with 74k gates [9], is too expensive (area, speed, power) for automotive applications. - The DSG has been optimized with respect to algorithmic strength, low power consumption, and small layout area. Although including some control logic and 40 bits of EEPROM, it measures only 1000 um x 750 um. Another interesting property of this DSG is, that it also incorporates an autonomous built-in self-test (BIST) block : A special challenge/key pair is chosen so that even 105 clock cycles are sufficient to ensure that every function's output in the DSG has toggled its output signal state at least once; even during this high node activity it takes only 9.3uA @ 5V operation. The chip is fabricated in a 1.2um CMOS process with EEPROM capability (SLM, DLP, n-well), carries about 14500 devices, and measures 2.75 mm x 2.0 mm.

Regarding software implementation, the DSG algorithm is slow. In contrast, during design of Rijndael the designers have considered both hardware and software applications. The same holds for the design of UICE: It shall be fast in software, even on small micro computers with only 8 bit data path.

2 Universal Immobilizer Crypto Engine (UICE)

Customers generally don't like proprietary algorithms delivered by suppliers, because they have to invest in system software and cannot switch easily from one supplier to another. For the supplier it seems to be a fine situation in the first view that the customer is somewhat bound to his product. However, this supplier cannot easily obtain larger market share, because the other customers are bound to their suppliers. – The same thoughts also apply to the question of second sourcing.

In order to overcome the 'trap' of proprietary algorithms the idea of a universal immobilizer crypto algorithm was born, an algorithm that has low complexity, is easy to understand, fast in software (compared to the DSG), and small in hardware.

2.1 UICE Architecture

The general block diagram of UICE is shown in figure 5. It is very similar to usual 8-bit micro computer architectures, but contains the specialized block S-BOX. This important S-BOX is **the non-linear function** needed for cryptographic algorithms [14,15,16]. Here, the application of an eight-by-eight S-Box [20,21] is proposed, i.e. the box has eight binary inputs and eight binary outputs. In software it is commonly presented as table of bytes with 256 entries – in hardware it could be implemented e.g. as a small ROM.

Everything is byte-oriented in order to speed up the cryptographic calculation on a 8-bit micro computer: the challenge register contains Nx bytes, the key register consists of Nk bytes, the accumulator (Accu) is also one byte wide, the arithmetic logic unit (ALU) operates on byte-operands and provides the designer with EXOR, ADD, SUB, AND, OR, etc. instructions. These instructions are normally one-cycle instructions (RISC principle) and executed quite fast.

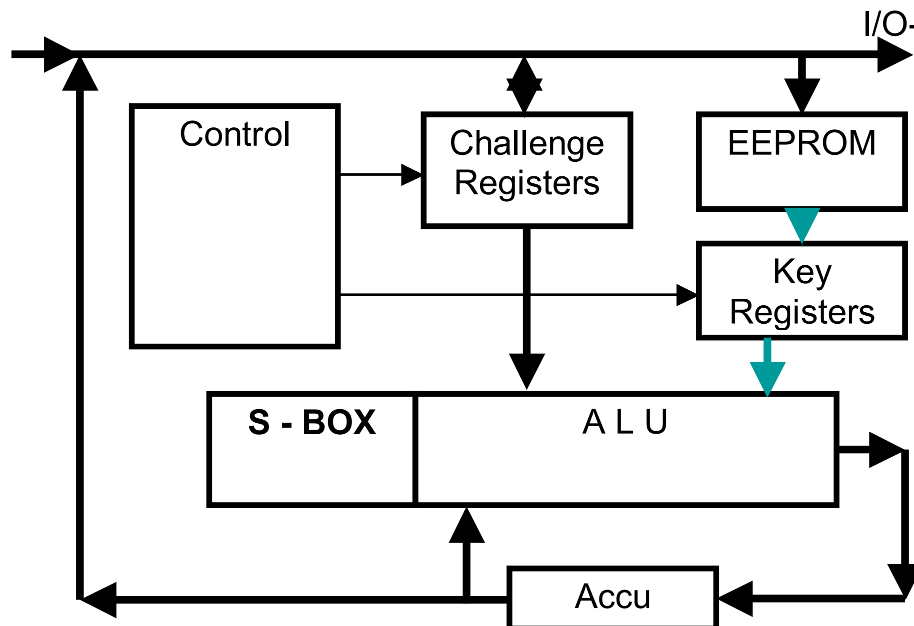


Figure 5 Block Diagram of the Universal Immobilizer Crypto Engine

The new algorithm has to provide confusion between key bits and challenge bits with respect to the final result that is stored in the challenge register. Diffusion can be performed by combining different challenge bytes with the previous result in the Accu. It is important to overwrite the initial challenge bytes as often as possible, for example Nr times, i.e. Nr rounds are performed with each of the challenge register bytes. Then the total time needed is proportional to Nx • Nr (see Table 1). -- During all rounds the key bytes have to be scheduled properly, i.e. the access to the key bytes must be distributed equally.

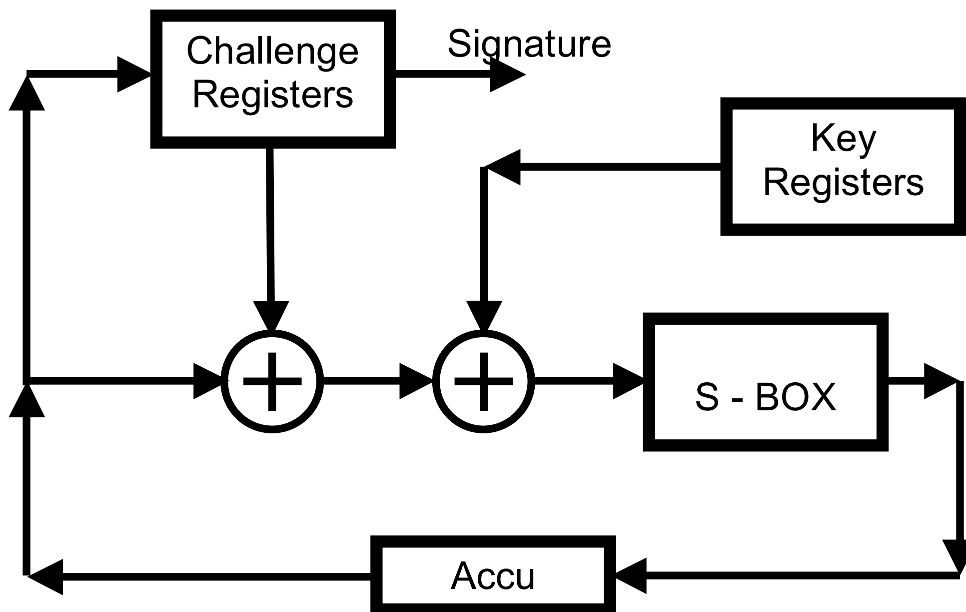


Figure 6 Detailed data flow in the UICE algorithm (first nine rounds)

Figure 6 shows the details of the data flow in the UICE core. One sub-round consists of the following operations:
a) Get Accu and challenge-byte
b) Perform EXOR
c) Get key-byte
d) Perform EXOR
e) Apply S-Box function
f) Store result in Accu
g) Update pointers to challenge register and key register banks.

2.2 UICE Speed

Although the content of the final S-Box has not to be defined in detail, the run times can already be determined, here (as presented in Table 1 below) by means of a notebook ASUS 7400 with Pentium 2, 400 MHz, Linux SuSE 8.0, gcc 2.95.3, and ANSI-C. It is impressive that UICE40 is 86 times faster than DSG, and that UICE128 is 19 times faster than AES128.

Table 1 Run Times of Different Cryptographic Algorithms

Algorithm	time us	Challenge Bytes	Key Bytes	Cipher rounds	Notes
DSG	480	5	5	10	{1}
UICE40	**5.6**	5	5	10	{2}
UICE64	**8.6**	8	8	10	{2}
UICE128	**8.6**	8	16	10	{2}
AES128	163	8	16	10	{3,4}[14]

Notes:
{1} DSG is based on many shift register operations, slow in software.
{2} UICE is based on Byte-Operations, is fast in software.
{3} AES (i.e. Rijndael) was chosen as proper compromise between hardware and software applications.
{4} AES128 is based on rijndael-alg-ref.c and needs three tables of 256 bytes each and also a table of 30 bytes [pulled from http://www.esat.kuleuven.ac.be/~rijmen/rijndael/index.html]. However, the code for key length 192 und 256 was eliminated. The sixteen bytes per block are filled with eight bytes of the Challenge, the rest is filled with zeroes. As result only eight bytes are taken.

Regarding the hardware implementation and the number of clocks required for the execution of the algorithm one can conclude:
- UICE40 needs only 50 clocks (5 bytes, 10 rounds)
- UICE64 and UICE128 need only 80 clocks (8 bytes, 10 rounds)
- AES needs 228 clocks on TMS320C6201 [17], 8390 clocks on 68HC08 [14], 14 clocks in a 173k gates VLSI [27], and one clock in a pipelined ASIC implementation [24].
  The number of clocks in a transponder chip depends strongly on the chosen implementation and may be estimated somewhere in between the values above due to the area/speed trade-off.

2.3 S-Box Considerations

The development of the S-Box is not a trivial task [21,23,26]. Of course, one could take directly the S-Box of the AES algorithm or its inverted S-Box [14]. However, we don’t know today, if there will not be found a short-cut for this function in the future [35].

An alternative approach is to search for a random S-Box with good differential distribution properties [15,18,20]. For the evaluation of this property a program was written that generates the differential distribution table ddt (see pseudo-code below). The S-Box provides good resistance against differential and linear attacks, if the counts in the ddt are low, e.g. only 2, 4 or 6. Note, that the ddt[0][0] must be excluded here, because that count adds always up to 256.

```
Integer  ddt[256][256] = 0;
Integer alpha, beta, x, xa;
For alpha = 0 to 255 do
  For beta = 0 to 255 do
    For  x = 0 to 255 do
      xa =  x  EXOR  alpha;
      If ( sbox[x]  EXOR sbox[xa] == beta )
            then  increment ddt[alpha][beta];
    Endfor;
  Endfor;
Endfor;
```

Table 2 shows the ddt values for two Rijndael s-boxes compared to random generated ones. Rijndael's S-Box contains exactly 255 times the value '4'; is this a strength or perhaps a weakness ? Cryptanalysts like to work on such challenging problems [35].

Because of the large search-space of **256!** a specialized search strategy is required. The related C-program is under development; examples 'random 2' and 'random 3' are listed below. The goal is to reach a maximum ddt value of 4, i.e. no single 6.

Table 2 Differential Distribution Table Results for Different S-Boxes

Name	CodeName	2	4	6	8	10	12
Rijndael	S	32130	**255**	0	0	0	0
Rijndael-Inv		32130	**255**	0	0	0	0
random 1	sboxAF	19763	4917	854	106	9	**2**
random 2		19661	4954	855	109	**14**	0
random 3	sbox130	22166	4629	400	**4**	0	0
(random->goal		28000	4000	**0**	0	0	0)

2.4 UICE Detailed Code

In the following the C-code of the current UICE algorithm is listed and discussed. It is containing the S-Box of 256 bytes, the pointers p1 and p2 that address the challenge register x[] and the key register k[] respectively. For the first nine rounds the key byte k[p2] is applied before the S-Box is used; the last round applies the k[p2] after the S-Box function was performed (compare Rijndael data flow [14], and the tutorial [25, page 5]).

The statements with copying of bytes at beginning and end of crunch_UICE are only needed for the test-bench which needs an input register and an output register separately (xin[], y[]). The parameter 'algo' is used in the test bench in order to select between UICE40, UICE64 and UICE128 versions. By means of '#define' directives the S-Box is selected and also the number of rounds. The modulo2 function is avoided in order to speed up especially the software implementation.

```
void crunch_UICE( unsigned char xin[], unsigned char y[],
                  unsigned char k[], int algo )
{
  int     round;   /* round counter */
  int     j;       /* inner counter */
  int     nx;      /* number of challenge bytes */
  int     nk;      /* number of key bytes */
  int     nrounds; /* number of rounds to run */
  int     p1;      /* pointer to actual challenge byte */
  int     p2;      /* pointer to actual key byte */
  unsigned char accu;
  unsigned char x[16]; /* Local working register for challenge and result */

  x[0] = xin[0]; /* Copy needed in order to */
  x[1] = xin[1]; /* avoid overwriting xin ! */
  x[2] = xin[2]; /* This is not needed in hardware. */
  x[3] = xin[3];
  x[4] = xin[4];
  nx = 5;
  nk = 5;
  if( algo == 4 || algo == 5 ) /* 8 bytes challenge */
  {
    x[5] = xin[5];
    x[6] = xin[6];
    x[7] = xin[7];
    nx = 8;
```

```
    nk = 8;
    if( algo == 5 ){ nk = 16;}
  }

#define SBOXx  sboxAF
#define SBOXy  sbox130
#define SBOX   S              /* selection of SBOX */
#define NR     10

  /* ----------------- start of algorithm ----------------- */
   nrounds =  NR;  /* number of rounds, i.e. outer loop count !! */
   accu = 0;
   p1   = 0;
   p2   = 0;

   for( round=1;  round <= nrounds-1; round++ )
   {
     for( j=1; j <= nx; j++ )
     {
       accu = accu ^ x[p1] ^ k[p2]; /* linear operation */
                                    /* key mixing before sbox operation */
       accu = SBOX[ accu ];         /* NON-LINEAR OPERATION */
       x[p1] = accu;                /* overwrite challenge byte ! */

       /* update the pointers */
       p1++;
       if( p1 >= nx ) p1 = 0;   /* 0..4 or  0..7 */

       p2 += 3;
       if( p2 >= nk ) p2 = p2 - nk;
     } /* for j */

         /* key scheduling so that x[] sees different k[] */
         if( round==2 ) p2++;
          else if( round==4 ) p2++;
           else if( round==6 ) p2++;
            else if( round==8 ) p2++;
        if( p2 >= nk ) p2 = p2 - nk;
   } /* for  round  */

   /* final round */
   for( j=1; j <= nx; j++ )
     {
       accu = accu ^ x[p1];     /* linear operation */
       accu = SBOX[ accu ];     /* NON-LINEAR OPERATION */
       accu = accu ^ k[p2];     /* key mixing after sbox operation */
       x[p1] = accu;            /* overwrite challenge byte ! */

       /* update the pointers */
       p1++;
       if( p1 >= nx ) p1 = 0;   /* 0..4 or  0..7 */
       p2 += 3;
       if( p2 >= nk ) p2 = p2 - nk;
      } /* for j */

  /* ----------------- end of algorithm ----------------- */

   y[0]=x[0]; /* Copy result to output; this is not needed in hardware! */
   y[1]=x[1];
   y[2]=x[2];
   y[3]=x[3];
   y[4]=x[4];
    if( algo == 4 || algo == 5 )
        {
           y[5] = x[5];
```

```
            y[6] = x[6];
            y[7] = x[7];
        }
} /* crunch_UICE -------------------------------------------------- */
```

2.5 UICE Statistical Tests

Statistical tests are important means in order to observe any weaknesses of the crypto-algorithm as early as possible during the development of the UICE algorithm. The following statistical tests are performed although the final S-Box has not been selected so far:

- Challenge Sensitivity, Key Sensitivity [Appendix B]
- Challenge Correlation, Key Correlation
- Challenge Affinity, Key Affinity
- FIPS 140-2 Random Number [19, 28]

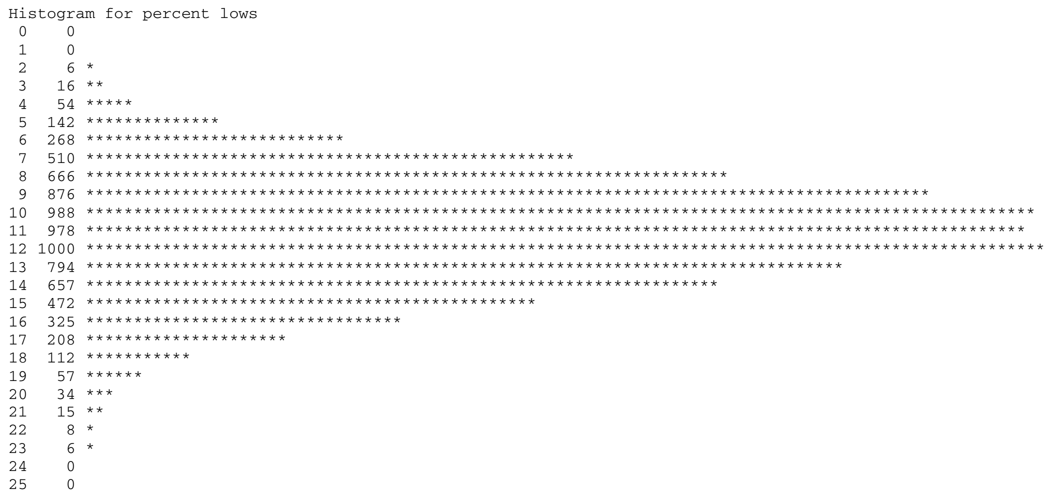
The current test results of the algorithm containing either two non-perfect S-Boxes or the AES S-Box [14] are promising. The test bench was executed 36 times: S-Boxes sboxAF, sbox130 (see appendix A.1 below) and S (Rijndael), round counts of 2, 3, 4, and 10, algorithms UICE40, UICE64 and UICE128.

The most critical test for the UICE128 algorithm is the key-sensitivity test [Appendix A.2], because 16 key-bytes have to be applied to eight challenge-bytes. The tests did not fail after three rounds demonstrating that the confusion and diffusion works well. Below the result after 10 rounds for the Rijndael S-Box:

```
k[] Key-Sensitivity Test

runs/experiment    =   50
experiments        =  100
significance level =   10%


Histogram for percent highs
 0    0
 1    1
 2    2
 3    7 *
 4   25 ***
 5   84 ********
 6  193 *******************
 7  330 *********************************
 8  536 ******************************************************
 9  722 ************************************************************************
10  943 **********************************************************************************************
11  939 **********************************************************************************************
12  976 **************************************************************************************************
13  906 *******************************************************************************************
14  765 *****************************************************************************
15  597 ************************************************************
16  434 *******************************************
17  314 *******************************
18  199 ********************
19   97 **********
20   55 ******
21   31 ***
22   24 **
23    6 *
24    4
25    2
```



```
Histogram for percent lows
 0    0
 1    0
 2    6 *
 3   16 **
 4   54 *****
 5  142 **************
 6  268 ***************************
 7  510 ***************************************************
 8  666 *******************************************************************
 9  876 ****************************************************************************************
10  988 ***************************************************************************************************
11  978 **************************************************************************************************
12 1000 ****************************************************************************************************
13  794 *******************************************************************************
14  657 ******************************************************************
15  472 ***********************************************
16  325 *********************************
17  208 *********************
18  112 ***********
19   57 ******
20   34 ***
21   15 **
22    8 *
23    6 *
24    0
25    0
```

This sensitivity test provides also a basis for the evaluation of the Strict Avalanche Criterion (SAC) [26]. Round after round the cipher text is checked how good the avalanche effect inside of the algorithm works, i.e. if the number of bits flipped compared to the original challenge is near to 50%. Figure 7 shows the result of the UICE tested in the version with 8 byte challenge and 16 byte key. After two rounds the 16 byte key has been applied the first time; this results in a probability of cipherbit change of less than 50%. But it is demonstrated that after three rounds the SAC has been fulfilled successfully.

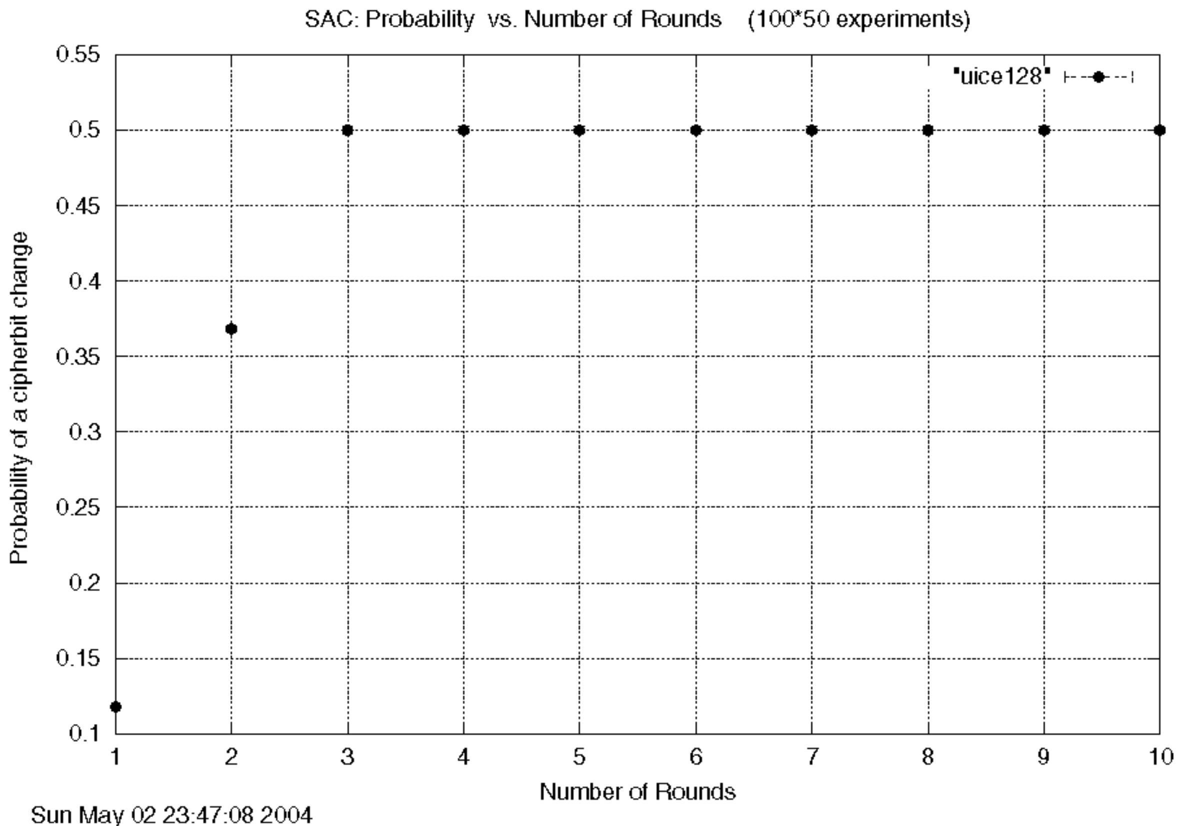


Figure 7 Strict Avalanche Criterion Test of UICE128 with 8 byte challenge and 16 byte key

2.6 Hardware Considerations

Since the UICE128 has not been built as ASIC design yet, an estimation of the necessary gate count will be performed. Assuming a digital library in 0.35um CMOS with NAND2 as one gate equivalent, D-FlipFlops with size of 4.3 gate equivalents, EXORs with 2.7 gate equivalents the following estimation is derived:

```
 64 bit challenge/state      64 x 4.3  = 275
128 bit key                 128 x 4.3  = 550
  8 bit accu                  8 x 4.3  =  34
 14 bit other DFF            14 x 4.3  =  60
---------------------------------------------
                            214 x 4.3  = 920
 16 x EXOR                   16 x 2.7  =  43
 SBOX  (max.)                            500
 Control/interface                       207
---------------------------------------------
                                        1670
```

The result is only 1670 gate equivalents for the UICE128 module; and it is even 275 gates less for the UICE64 variant. Such a small cryptographic module allows the application in embedded systems such as SoC designs. Examples for applications are e.g.:
- Bus encryption between modules in smart cards,
- Input/output encryption between different integrated circuits on a board,
- Authentication process before a Flash memory can be readout,
- Encrypt data in Flash memory or ROM, decrypt in real-time before it is used in the micro-processor
- Generation of pseudo random numbers (PRNG),
- Postprocessor for a True Random Number Generator,
- etc.

3 Conclusions

The principle of an immobilizer and its battery-less RFID operation has been described. The transponder not only contains an analog-front-end for the selected operating frequency, logic array and EEPROM memory, but also a cryptographic co-processor module. Depending on the requirements such a module is either a custom design or VHDL-based gate array, is proprietary or standardized algorithm, is high-speed or very-low-power.

In order to overcome the problems with proprietary algorithms, the concept of Universal Immobilizer Crypto Engine (UICE) has been developed. This non-proprietary algorithm has been tailored to the architectures of 8-bit microprocessors so, that it is fast in software and in hardware. The minimum hardware requirements are: 8 bytes for the challenge register, 8 or 16 bytes for the key register, one byte for the Accu, an 8-bit EXOR function, a 256-byte ROM and a small control logic with two counters and two pointers to the registers; the challenge register holds the final result.

The statistical tests performed so far are very promising. The Accu is performing the diffusion task well despite chaining only eight bits from one sub-round to the next one. Also the non-perfect randomly-selected S-Box is sufficient. After ten rounds the UICE seems to be strong enough to withstand differential and linear attacks [22]. Furthermore, the FIPS 140-2 Random Number Generator tests performed over 20000 result bits do not show any weaknesses of UICE in counter mode.

Side-Channel Attacks such as differential power analysis (DPA) [31,32], electromagnetic radiation analysis etc. are not covered here. These problems have to be worked on during implementation of UICE on the SoC or transponder chip.

Table 3 compares some important properties of UICE versus DSG and AES. It is recommended that industry adopts this free 'pseudo-standard' concept for the next immobilizer generation because of its low-complexity, sufficient cryptographic strength and high speed in hardware and software.

Table 3 Comparison of Different Algorithms

Property / Name	UICE128	DSG	AES
Complexity	low	high	medium
Speed Software	high	low	medium
Speed Hardware	high	medium	medium
Chip Area	low	medium	medium
Security	higher [33]	high	highest
Usage	signature	signature	mass data
Configuration	ECB	ECB	ECB, CBC, etc.
Application	immobilizer and more	immobilizer	data encryption

4 References

[1] J. Schuermann, H. Meier, TIRIS - Leader in radio frequency identification technology, Texas Instruments Technical Journal, TITJ Vol.10, No. 6, Nov. 1993, pp. 2-14
[2] U. Kaiser, W. Steinhagen, A low power transponder IC for high performance identification systems, Proceedings of CICC'94, San Diego, CA, USA, May 1-4, 1994, pp. 14.4.1-14.4.4
[3] W. Steinhagen, U. Kaiser, A low power read/write transponder IC for high performance identification systems, Proceedings of ESSCIRC'94, Ulm, Germany, September 20-22, 1994, pp. 256-259
[4] U. Kaiser, W. Steinhagen, A low power transponder IC for high performance identification systems, IEEE Journal of Solid-State Circuits, VOL. JSSC 30, March 1995, pp. 306-310
[5] J. Gordon, U. Kaiser, A. Sabetti, A Low Cost Transponder for High Security Vehicle Immobilizers, Proceedings of ISATA, Florence, Italy, 3.-6.June.1996, Automotive Electronics, 96AE001
[6] J. Gordon, Designing codes for vehicle remote security systems, Concept Laboratories Ltd. and Police Scientific Development Branch, Herfordshire, G.B., 1994, pp. 1-22
[7] B. Schneier, Applied cryptography, Addison Wesley, 1995
[8] D. Stinson, Cryptography, Theory and Practice, CRC Press, 1995
[9] C. Yang, T. Chang, C. Jen, A new RSA cryptosystem hardware design based on Montgomery's algorithm, IEEE Transactions on Circuits and Systems II, Vol. 45, No. 7, July 1998, pp. 908-913
[10] A. Scheerhorn, DSG Algorithm Evaluation, EVAL11.DOC, CCI, Meppen, 2.Dec.1997
[11] U. Kaiser, A Low-Power Digital Signature Transponder IC for High Performance RFID Authentication, Proceedings of European Conference on Circuit Theory and Design, ECCTD'99, Stresa, Italy, Aug. 29 - Sep. 02, 1999, pp. 45-48
[12] U. Kaiser, Theft Protection by means of Embedded Encryption in RFID-Transponders (Immobilizer), ESCAR - Embedded IT-Security in Cars, Bochum, Germany, 18-19.Nov.2003
[13] S. Sarma, S. Weis, D. Engels, RFID and Security and Privacy Implications, CHES 2002, LNCS 2523, Springer 2003, pp. 454-469
[14] J. Daemen, V. Rijmen, AES Proposal: Rijndael, Version 2, 03/09/99, 45 pages and related Reference Code in C http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaelref.zip
[15] J. Seberry, X. Zhang, Y. Zheng, Pitfalls in Designing Substitution Boxes, S-Box, Crypto'94, Aug.1994, pp 383ff
[16] A. Menezes, P. van Oorshot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997
[17] T. Wollinger, M. Wang, J. Guajardo, C. Paar, How Well are High-End DSPs Suited for the AES Algorithms ? - AES Algorithms on the TMS320C6x DSP, The Third Advance Encryption Standard (AES3) Candidate Conference, April 2000, New York, 11 pages

[18] T. Ritter, S-Box design: A Literature Survey, Research Comments, http://www.ciphersbyritter.com/RES/SBOXDESN.HTM
[19] NIST FIPS 140-2, Security Requirements for Cryptographic Modules, May 25, 2001, http://csrc.nist.gov/cryptval and http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
[20] J. Xu, H. Heys, A New Criterion for the Design of 8 x 8 S-Boxes in Private-Key Ciphers, IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'97), May 1997
[21] J. Gordon, A. Retkin, Are Big S-Boxes Best ?, IEEE Workshop on Communication Security, Santa Barbara, Cal. 1981, pp. 1-6
[22] H. Heys, S. Tavares, Substitution-Permutation Network Resistant to Differential and Linear Cryptanalysis, Journal of Cryptology, Vol. 9, No. 1, pp.1-19, 1996
[23] Webster, Tavares, On the Design of S-Boxes, Proc. CRYPTO 1985, LNCS 218, Springer, 1986, pp.523-534
[24] J. Rejeb, V. Ramaswamy, K. Ghadiri, Hardware Implementation of the Rijndael Algorithm for High-Speed Networks, ISPC 2003, March 2003, Dallas, 6 pages
[25] H. Heys, A Tutorial on Linear and Differential Cryptanalysis, Technical Report CORR 2001-17, Mar. 2001, http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.ps
[26] H. Heys, S. Tavares, On the Design of Secure Block Ciphers, Queen's 17th Biennial Symposium on Communications, Kingston, Ontario, May 1994, 6 pages
[27] H. Kuo, I. Verbauwhede, Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm, CHES 2001, LNCS 2162, pp. 51-64, Springer 2001
[28] National Institute of Standards and Technology, FIPS PUB 140-2 Annex A: Approved Security Functions, www.nist.gov/cmvp.
[29] SpeedPass web site, http://www.speedpass.com
[30] C. Kern, RFID-Technology – Recent Development and Future Requirements, Proceedings of European Conference on Circuit Theory and Design, ECCTD'99, Stresa, Italy, Aug.29-Sep.02, 1999, pp. 25-28
[31] Kocher, Jaffe, Jun, Differential Power Analysis, Advances in Cryptology, CRYPTO'99, LNCS 1666, Springer 1999, 10 pages
[32] Kocher, Evaluating Cryptosystems, 31 slides, Cryptography Research 2002, http://www.cryptography.com/resources/whitepapers/HackingCryptosystems.pdf
[33] M. Loney, Moore's Law is the biggest thread to privacy, according to Phil Zimmermann, news.zdnet.co.uk and www.silicon.com, 29.Apr.2003
[34] UICE slides, AES4, May 2004, Bonn http://www.aes4.org/english/events/aes4/downloads/AES4_UICE_slides.pdf
[35] N. Courtois, How Fast can be Algebraic Attacks on Block Ciphers ? , May 2006, http://eprint.iacr.org/2006/168

Appendix

A.1.1 Random S-Box beginning with 0xAF

```
unsigned char sboxAF[256] =
 {0xAF,0x1B,0xDD,0xBC,0x30,0xEB,0xF0,0x56,0xC1,0x08,0x93,0x36,0x03,0xCB,
  0x81,0x80,0x43,0x2F,0xDF,0x2D,0x26,0x05,0x0A,0xF8,0x7D,0x21,0xE0,0xC4,
  0x06,0xD5,0xA6,0xE8,0x8E,0x70,0xDC,0xA4,0x6D,0x23,0xAC,0x18,0x40,0x00,
  0x64,0x0E,0xF6,0x79,0xB5,0x1F,0x5D,0x9A,0x3B,0xFA,0x48,0x5F,0x74,0xA1,
  0x8D,0xD3,0x5C,0x4E,0x9E,0x14,0x25,0xEF,0xD6,0xC9,0x3F,0xC5,0xA0,0x10,
  0x50,0xFB,0x31,0xF4,0x17,0x88,0xAB,0x32,0x76,0x3E,0x15,0x2A,0x3D,0xA9,
  0x52,0x20,0xC3,0xFC,0x7B,0x49,0x3A,0x6E,0xB7,0x1D,0xAD,0xAA,0x5A,0x0D,
  0x35,0x38,0xC8,0xF5,0xF3,0xB3,0x8F,0xE6,0x13,0x55,0x33,0x8A,0xC0,0x67,
  0x2E,0xE7,0x82,0x8C,0x09,0xCF,0x1E,0x97,0x28,0x07,0xBA,0x4D,0x42,0x04,
  0x73,0x41,0x5B,0xB1,0xF9,0xE5,0xF7,0x6C,0xD8,0x12,0x8B,0x84,0xCC,0xB0,
  0x69,0x37,0xAE,0x6F,0xE2,0xDB,0x0C,0x86,0x29,0x78,0x34,0x7C,0x1A,0x85,
  0x27,0xA3,0x9B,0x92,0xE3,0xBD,0x59,0x63,0x66,0x19,0xCA,0x5E,0xFF,0x75,
  0x72,0x24,0x4F,0x47,0x61,0x11,0x0B,0xBE,0xA7,0x16,0x3C,0xB2,0xFD,0x7F,
  0x44,0x99,0x6B,0x98,0x22,0x46,0x4A,0x1C,0x02,0x6A,0x51,0x39,0x60,0x4B,
  0x57,0x01,0x2C,0xE1,0xEE,0x83,0x89,0xDA,0x58,0x0F,0xBB,0x2B,0xD2,0xD4,
  0x62,0x9F,0x90,0x7E,0xDE,0xB8,0x4C,0xCD,0x68,0xA8,0xF2,0x54,0xE9,0xE4,
  0xF1,0xEA,0xD7,0x77,0x9D,0x96,0xEC,0xFE,0xB9,0x91,0xBF,0xD1,0xD9,0xA2,
  0x95,0xED,0xA5,0xC2,0x7A,0xC6,0xC7,0x45,0x94,0xB6,0x65,0xD0,0xCE,0x9C,
  0x71,0x87,0xB4,0x53 }; /* ddt: 19763 4917 854 106 9 2 0 */
```

A.1.2 Random S-Box beginning with 130

```
unsigned char sbox130[256] =
{130,150,219,161,127,160,229,198, 99, 26, 22, 63, 74,136,215,201
, 82,195,  3,225,239, 94,129, 80, 18,213,149,245,  7, 57,197,115
,113,230,116,163,212,133,162,222,105, 60, 19,170,244, 30,  1,137
,176, 27,185, 42,153, 16,104,202,221, 11,172,190,154,151,103, 71
, 28,187,  2, 88,231,204, 17, 37,228, 73, 44, 31,134,100,144,211
, 89,117,108, 39,227,241,232,247, 20,226,110,254,169, 14,174,119
,131,152, 55,205, 49,164,142, 43,132, 12, 98,224,135,157,114, 83
, 78,236,111, 23,147, 70, 47,252,189, 32, 41,124,120, 58,102, 33
,234,184,158,177,140,121,246,180,175, 45,101,233,168,138,203,188
, 15, 97,183,196, 59,250, 54,  6,148,207, 72,118,206, 86, 48,112
,199, 36, 96,145, 75, 85,220,186,106,217, 34,240, 87,178, 69, 65
,238, 91,179,159,249,146,107,214, 56,141, 10,243,125,165,  8, 29
, 52, 13,253,193, 66, 21,126, 50, 77,251,143, 84,192, 25,  9, 79
, 93, 46, 81,  0, 38,  4, 35,  5,194, 95,128,242,122,156,109, 90
,248,166, 61, 76,167, 67,210,155,139,255, 24, 40, 53, 62,216,218
,173,181,200, 68,191,223,171, 64,209, 92,123,237,182,235, 51,208
}; /* ddt: 22166 4629 400 4 0 0 0 0 */
```

A.2 Part of the test bench (e.g. sensitivity test)

```
Tries to find sensitivities between input bit and output bit,
i.e. check if avalanche effect is sufficient,

  a) between key bit and response bit (described below)
  b) between challenge bit and response bit

Details:

 using:   crunch( challenge, response, key );
```

```
n = 50;
 Do for 100 experiments

    Do for n runs

       /* setup  random challenge  and  random key  */
       for( j=0; j<CHALLENGEBYTES; j++ )   x[j] = RandByte();

       for( j=0; j<KEYBYTES; j++ )         k[j] = RandByte();

       /* find yy[] as a reference */
       crunch( x, yy, k );

       for all KEYBITS do

         toggle  k [ KEYBIT ];
         crunch( x, y, k );
          /* y and yy should be different ! */

         for all RESPONSEBITS do
           if( y[ RESPONSEBIT ] == yy[ RESPONSEBIT ] )
             then increment hist[ RESPONSEBIT ][ KEYBIT ];
         end
         toggle  k [ KEYBIT ]; //restore k
       end
    end

    Find HIGHs and LOWs in histograms

 end

 Print final results.


Find HIGHs and LOWs in histograms
---------------------------------
#define CHISQ_90 0.0157908
#define CHISQ_10 2.70554
CHISQ_LO = CHISQ_90;
CHISQ_HI = CHISQ_10;

do for all key bits        // CHI**2 Test
  do for all response bits
  chisq = ( 2*hist[i] - n );
  chisq = chisq*chisq / n;
  if( chisq>CHISQ_HI ) highs[i]++;
  if( chisq<CHISQ_LO ) lows[i]++;
  hist[i] = 0;
  end
end


Print final results
-------------------
print for every response bit  the HIGHs / number of experiments in %;
print for every response bit  the  Lows / number of experiments in %.
Horizontal: response bit position
Vertical  : key bit position

print histograms
Expected result: about 10% for every bit position
```