Security Upgrade of Existing ISDN Devices by Using Reconfigurable Logic

Hagen Ploog, Mathias Schmalisch, Dirk Timmermann

Department of Electrical Engineering and Information Technology, University of Rostock
Richard-Wagner-Str. 31, 18119 Rostock, Germany
hp@e-technik.uni-rostock.de

Abstract. Integrated Services Digital Network (ISDN) is a de facto worldwide standard for wired digital telephony. Since ISDN was developed in 1984 as a public network it does not support modern sophisticated fraud detection systems. Indeed most current installations of ISDN are incapable of supporting encrypted transmission. Securing user information is therefore normally done outside the ISDN communication environment. In this paper we present our experience in using reconfigurable logic for securing existing ISDN devices against intruders in point to point communication.

1 Introduction

Many companies are using ISDN-based solutions for intranet or tele-applications, such as, tele-working or tele-conferencing. Since ISDN itself does not support secure transmission of data (phone, fax) securing user information is therefore normally done outside the ISDN communication environment. It can be realized either by dedicated hardware or a software package running on the data providing unit. The latter is not recommended for reasons of mobility since it is unwieldy to boot a computer just to make a secure phone call.

In this paper we present our experience in using reconfigurable logic for securing existing ISDN adapters against intruders in point to point communication. Instead of developing a completely new security device we expand the services of an existing least cost router (LCR) with a daughterboard containing a FPGA for speeding up the cryptographic processes. To ease the design, we focus on the implementation of the security device, so we do not reimplement the LCR-functionality. ISDN offers two B-channels for data transmission and one D-channel to control the communication. For secure phoning only one B-channel is used, but with different keys in each direction.

The paper is organized as follows: We briefly describe the structure of SECOM in chapter 2. In chapter 3 we focus on the protocol for key-exchange. In chapter 4 we present some implementation issues of the cryptographic algorithm. We finally close with the conclusion in chapter 6.

2 Architectural description

The LCR we used contains a V25 microprocessor, RAM, EPROM and the complete ISDN-interface. To update the LCR and to reduce the number of additional wires to connect the board with the LCR, we built a piggy-pack board containing the FPGA (XC4020), boot-EPROM, RAM, some glue logic and the replaced EPROM of the LCR.

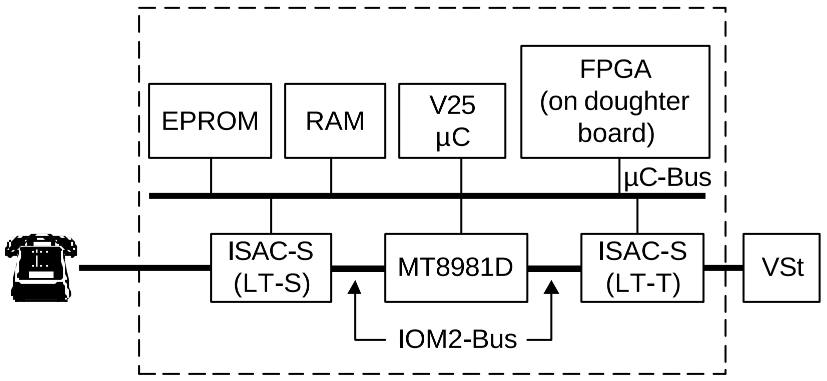


Fig. 1. Block diagram of the expanded LCR.

Generating a secure phone call is a three step process. During the first step we autodetect the existence of a corresponding encryption device. The second step is to exchange the session keys between both subscribers. The users input data is then encrypted and transmitted throughout the third step.

Autodetection is realized by transmitting a four byte sequence (0x00, 0x01, 0x02, 0x03) from the caller to the receiver. The probability that this sequence exists during regular communication is 2^{-32}, but it is only used during start up and can be easily expanded and modified for safety reasons. If the receiver detects this sequence, it also starts transmitting the same string several times to signal the existence of an encryption device. This process of repeating data sequences is often referred to as bit reiteration.

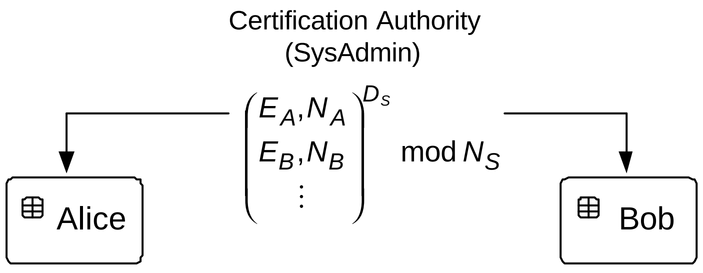
The controlling microprocessor decides whether the key exchange was successful in generating an optical feedback to the user and start the transmission of the encrypted session keys.

Since key exchange and data encryption are time-independent, we use the FPGA twice, but with different contents. Every time a phone call is going to be initiated the FPGA boots with an implementation of the public-key algorithm. After successfully encoding the session keys the FPGA is rebooted with the private-key algorithm.

3 Basic Security Model

We are using a hybrid system to secure telephone calls. A public key algorithm is used to encrypt the session key. The encrypted key is then transmitted via an insecure ISDN-channel to the second subscriber, where it will be decrypted. The received key is used to decrypt the user data bytes with a fast private key algorithm. It is necessary to use a new session key each time, since static keys could be broken with an minor amount of time (and money) and hence the whole system would become insecure.

Using a hybrid system normally requires so called trustcenters for authorization of both sides' public keys, since both subscribers wants to be sure that the received key belongs to each other and not to an intruder. Planning our system just as a small point to point communication tool, we could replace the functionality of the trustcenter to an indoor operator, who is only signing the public keys. Therefore, an intruder can spy out the public keys of all subscribers but he is not able to generate a new list. Throughout it's usage, the public keys were read from a smart card, which is initialized by the operator. Alice, as the caller is signing the session key with her private key for authorization. She then encrypts the authorized key with Bobs public key to insure that only Bob can decipher the session key.



Certification Authority
(SysAdmin)
$\begin{pmatrix} E_A, N_A \\ E_B, N_B \\ \vdots \end{pmatrix}^{D_S} \bmod N_S$
Alice
Bob

$$c_{A1} = (key_A, IV_A)^{D_A} \bmod N_A \qquad c_{B1} = (key_B, IV_B)^{D_B} \bmod N_B$$

$$c_{A2} = \left(c_{A1}, N^{\underline{o}}\right)^{E_B} \bmod N_B \qquad c_{B2} = \left(c_{B1}, N^{\underline{o}}\right)^{E_A} \bmod N_A$$

$$\xrightarrow{c_{A2}} \qquad \xleftarrow{c_{B2}}$$

$$m_{A2} = (c_{B2})^{D_A} \bmod N_A \qquad m_{B2} = (c_{A2})^{D_B} \bmod N_B$$

$$(key_B, IV_B) = (m_{A2})^{E_B} \bmod N_B \qquad (key_A, IV_A) = (m_{B2})^{E_A} \bmod N_A$$

$$\xrightarrow{c_{Ai}} \qquad \xleftarrow{c_{Bi}}$$

$$c_{Ai} = d_{Ai} \otimes r_{Ai} \qquad c_{Bi} = d_{Bi} \otimes r_{Bi}$$

$$d_{Bi} = c_{Bi} \otimes r_{Bi} \qquad d_{Ai} = c_{Ai} \otimes r_{Ai}$$

Fig. 2. Implemented protocol for key-exchange between two subscribers

4 Implementation issues

4.1 Random numbers

Some non time critical tasks such as the generation of random numbers can be done by the microcontroller during off line time. The system requires three 64 bit numbers for the triple DES core and one 64 bit number for the initialization vector (IV). These numbers are used as the session keys for the next phone call. We use a linear congruence generator for generating the numbers corresponding to

$$X_n = (2^{34} + 1) \bullet x_{n-1} + 1 \bmod 2^{64} \qquad (1)$$

These keys each are encrypted inside the security device with it’s private key.

4.2 Modular exponentiation

We use the RSA algorithm [1] for key exchange. The most time consuming operation in the RSA algorithm is the modular exponentiation (ME) of long integers. To guarantee security, significantly greater wordlengths than in conventional computers are used (up to 2048 bits). In the RSA algorithm the public key consists of an exponent E and the system modulus N as the product of two large primes p and q, $N = p \cdot q$. The private key consists of the same modulus N and an exponent D fulfilling $D = E^{-1} \bmod (p\text{-}1) \cdot (q\text{-}1)$. The public key E and N can be released as the system modulus. The exponent D has to be hidden since it is the secret key of the cryptosystem.

To encrypt the message m one has to compute $c = m^E \bmod N$. Decryption of the cipher c is done by computing $m = c^D \bmod N$. The generation of the parameters (p, q, N, E, D) is software based and realized by the controlling microprocessor once during initialization time. The generated public key E and the modulus N are transmitted to the certification authority to get signed. Since encryption and decryption are the same operation but with different exponents, RSA can be used for authentication and ciphering.

Let n the number of bits representing the modulus N. A widely used method to perform modular exponentiation is the 'square and multiply'-technique. To compute $b^E \bmod N$, the exponentiation can be split into a series of modular multiplications, where b, E, and N are three n-bit non negative integers related by $b, E \in [0, N\text{-}1]$.

```
Algorithm 1
Input : b, E, N; 0 < b, E < N
Output: b^E mod N
Y = 1
For i = (n-1) down to 0 loop
   Y = Y * Y mod N
   Y = Y * B mod N if ($E_i$ == 1)
```

Unlikely, due to the correlation of E and D, we do not profit by using short exponents during the decryption process so the algorithm takes 1.5 n modular multiplications in the average and 2 n in the worst case.

In 1985, P. L. Montgomery proposed an algorithm for modular multiplication $A \cdot B$ mod N without trial division [2].

Montgomery multiplication Let A, B be elements of Z_N, where Z_N is the set of integers between [0, N-1]. Let R be an integer relatively prime to N, e.g. gcd(R, N)=1, and $R > N$. Then the Montgomery algorithm computes

$$\text{MonProd}(A, B) = A \cdot B \cdot R^{-1} \bmod N. \qquad (\mathbf{2})$$

If R is a power of 2 the Montgomery algorithm performs a division by a simple shift, but it is working with any R being coprime to N.
For the computation of the Montgomery product we need an additional value N' satisfying $1 < N' < R$ and $N' = -N^{-1}$ (mod R). The algorithm for MonProd is given below:

```
Algorithm 2
Input :  $\bar{a}, \bar{b}, R^{-1}, N, N'$
Output:  $\bar{a} \cdot \bar{b} \cdot R^{-1} \bmod N$
$t = \bar{a} \cdot \bar{b}$
$m = t\,N' \bmod R$
$u = (t + m \cdot N) / R$
if $u \geq N$ then $u = u - N$
```

MonProd using multiple-precision arithmetic To avoid brute force breaking of the cryptosystem the length of N should be at least 512 bits or more.

In multiple-precision (MP) arithmetic, large integers X, with x representing the length of X in bits (x=512, 768, or 1024 bits) are broken down into s quantities of length w, e.g., w=8, 16 or 32 bits. The arithmetic operations are implemented recursively, increasing the index from 0 to (s-1), with $s = \lceil x/w \rceil$. Any x-bit integer X can be interpreted as:

$$X = \sum_{i=0}^{x-1} x_i 2^i = \sum_{i=0}^{s-1} d_i (2^w)^i = \sum_{i=0}^{s-1} d_i W^i \qquad (\mathbf{3})$$

If $w \cdot s > x$ then X has to be padded with zeros on the left.

Dussè and Kalinski first noted that in the case of multiple-precision it is possible to use $N_0' = -N_0^{-1} \bmod W$ (N_0' and N_0 are the w least significant bits of N' and N' respectively) instead of N'. As a consequence, algorithm 2 can be written in the case of multiple precision as [3]:

```
Algorithm 3
Input : $\bar{a}, \bar{b}, W, N_0'$
Output: $\bar{a} \cdot \bar{b} \cdot R^{-s \cdot w} \bmod N$
t=0
n‘_t = N‘[0]
for i = 0 to s-1
  for j = 0 to i-1
    t = t + a[i-j]*b[j]
    t = t + n[i-j]*m[j]
  endfor
  t = t + a[0]*b[i]
  m[i] = t[0]*n‘_t mod W
  t = t + m[i]*n[0]
  SHR(t,w)
endfor
for i = s to 2s-1
  for j = i-s+1 to s-1
    t = t + a[i-j]*b[j]
    t = t + n[i-j]*m[j]
  endfor
  m[i-s] = t[0]
  SHR(t,w)
endfor
```

The result will be found in *t*.

As shown here, the multiplication and accumulation are the heart of the algorithm. In each step *j* we have to compute $acc = acc + g \cdot h$, with *g* equals *a* or *n* and *h* equals *b* or *m*, depending on if we are in the multiplication or reduction part of the algorithm. It can be shown that the Montgomery multiplication can be performed without precalculation of N_0' [4]. This is achieved by interleaving the multiplication with the accumulation, e.g. $t_j = t_{j-1} + g \cdot h_j$ and some modification on the LSB.

A VHDL description of an arithmetic co-processor for computing modular exponentiation based on the optimized Montgomery multiplication is given in [6]. This model can be parameterized by *w* and *N*. Table 1 shows the required time to perform a modular exponentiation in milliseconds (f=16 MHz). In our implementation (n=256, W=16) the deciphering of the key is done in ~37 ms.

Table 1. Time [ms] for computing a modular exponentiation

w	$\log_2(n)$		
	256	512	1024
8	48.96	392.4	3143
16	12.24	98.1	785.6
32	3.06	24.5	196.4

4.3 Secure phone calls

After successfully decoding the session keys, the FPGA is re-booted with the private-key algorithm. Therefore the FPGA is reset and a dedicated IO-port controlled by the microprocessor selects the boot-area. As phone calls require real-time encryption we implement the block cipher in the so called cipher feedback mode (CFB) to achieve fast ciphering. Encryption in the CFB-mode is achieved by XOR-ing the plaintext with the m bit output of a key stream generator and feeding back the resulting cipher into a n bit shift register which was initialized with the initialization vector (IV).

Decryption is again a simple XOR-ing with the same output of the key stream generator since A = A xor B xor B. In that way the block cipher is used as a pseudo-random number generator. Any block cipher could be used to overcome export restrictions without changing the principle cryptographic process.

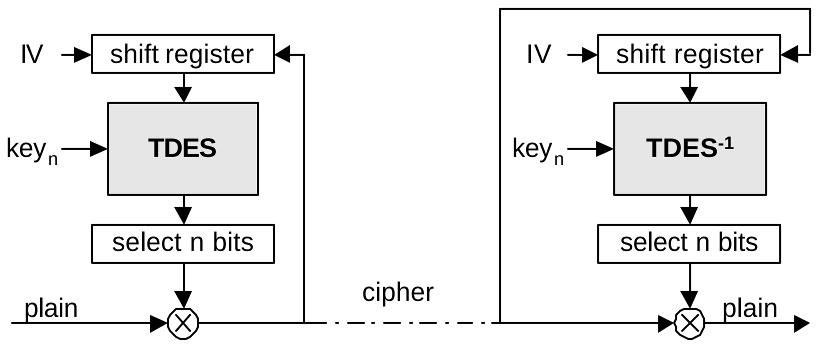


Fig. 3. CFB-mode

Besides real time encryption, the CFB-mode realizes self-synchronization which minimizes communication overhead. If a synchronization error occurs by corruption of the cipher during transmission, the decrypting unit generates wrong plaintext as long as the modified cipher is in the input shift register. For the case of telephony, synchronization is recovered after

$$\frac{n\,bit\ input\ register}{m\,bit\ \ feedback \cdot fclk} = \frac{64\,bits}{8\,bits \cdot 8\,kHz} = 1\,ms \tag{4}$$

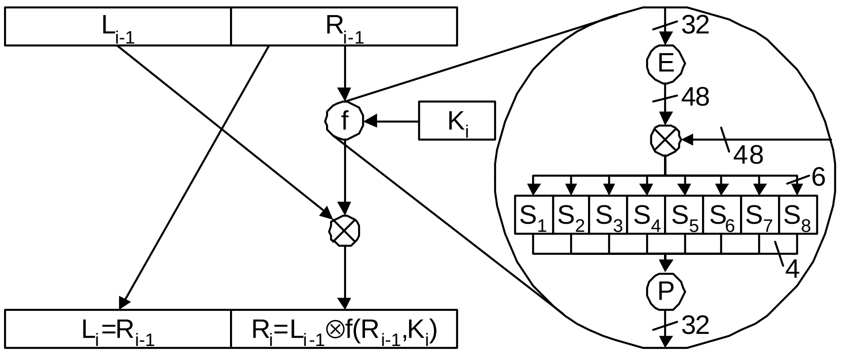
Unfortunately the CFB-mode enables intruders to insert faulty data into the stream without being detected if the data was taken from the actual call. So the intruder can retransmit recorded data, but since he is not able to decipher the transmitted cipher in real time he does not know *what* he is transmitting.

If someone tries to retransmit recorded data in a later phone call, the receiver just hears noise, since the session-key was changed.

4.4 DES implementation

For the block cipher we use the Data Encryption Standard (DES) algorithm [5] with an expansion to triple DES to encrypt the user's data. In triple DES data is encrypted with the first key, decrypted with a second key and then encrypted again with a third key. In the DES algorithm the incoming data and key are passed through input and output permutations. Between these permutations the data is passed 16 times through a Feistel network, while 16 keys were generated simultaneously.

The data is split into two parts, L and R. R_{i-1} is modified by the function f and XORed again afterwards with L_{i-1} to build the new R_i. R_{i-1} is directly passed through to L_i. Fig. 4 is showing one round of the DES algorithm.



L_{i-1}
R_{i-1}
f
K_i
$L_i=R_{i-1}$
$R_i=L_{i-1}\otimes f(R_{i-1},K_i)$
32
E
48
48
6
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
4
P
32

Fig. 4. One round of DES

During the f-function data is expanded and XORed with the sub-key k_i. The received value is fed through eight S-boxes, which are a simple table lookups. One S-box substitution is shown in Fig. 5. Each S-box takes about 10 configurable logic blocks (CLBs), eight for the LUTs and two for the mux.

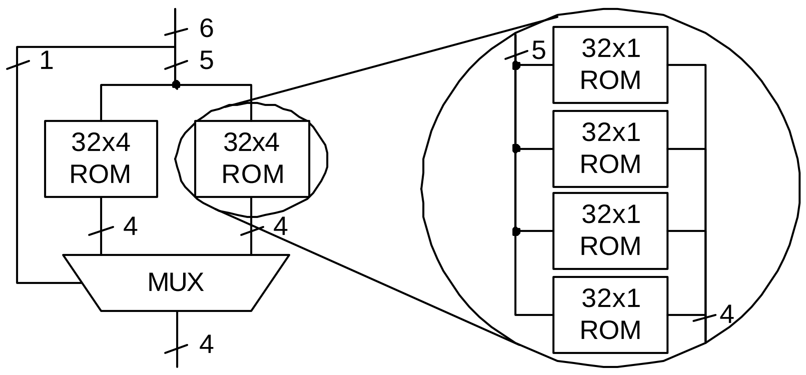


Fig. 5. Implementation of one S-box

Since ISDN clock-frequency is relatively low (8 KHz), we do not profit from loop unrolling, so we implemented an iterative version of the DES-algorithm, which leads consequently to a smaller architecture. The structure of the block diagram is shown in Fig. 6.

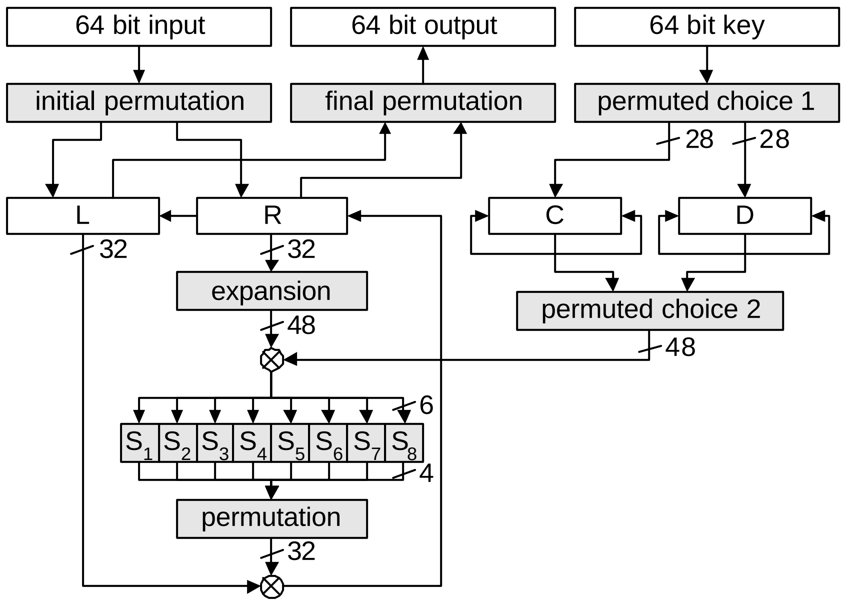


Fig. 6. Looped implementation of DES

It is widely known that the standard DES algorithm was broken with the help of the internet several months ago [7]. Therefore, we expand our core from DES to triple DES by adding a key look-up table (LUT). This LUT can be written by the microprocessor to update the session key.

During online encoding, raw data is written into a dedicated address register belonging to the address space of the FPGA. The data is XORed inside the FPGA and moved to the same register again. Reading from that register automatically starts the controlling finite state machine to generate the next random number. Therefore key_1, key_2 and key_3 are successively read out of the LUT and fed into the DES core, where data is encrypted/decrypted with the key.

The implemented DES core takes 17 clock cycles to perform the DES algorithm. The modified version takes 51 clock cycles and three extra cycles to store the data during the reading of the keys from the LUT. Therefore, the generation of a new cryptographic secure random number takes 3.2 µs.

5 Conclusions and future work

We have shown that it is possible to protect ISDN communication. By the use of reconfigurable logic it is possible to implement different algorithms as the two we have used.

We use a small RSA key, because the encoding is fast. In order to increase the security during the key exchange, it is necessary to expand the RSA algorithm to 1024 bit or more. Therefore, a wider RAM has to be implemented. For fast encoding of such bit widths, it is necessary to use a larger FPGA to implement a larger multiplier for Montgomery multiplication. By the usage of a larger FPGA it is also possible to secure the second B-channel for communication.

It is also necessary to update the smart cards with new signed keys without distributing the smart cards manually by the certification authority.

6 Acknowledgements

We are very grateful to Dieter Fischer from TAS GmbH & Co KG Mönchengladbach, Germany, for making us a LCR prototype board available for the development of the ISDN encryption device.

7 Reference

[1] Rivest, R.L., Shamir, A., Adleman, L.: A Method of obtaining digital signature and public key cryptosystems, Comm. Of ACM, Vol.21, No.2, pp.120-146, Feb.1978

[2] Montgomery, P.L.: Modular Multiplication without Trial Division, Mathematics of Computation, Vol.44, No.170, pp.519-521, 1985.

[3] Dusse, S.R, Kaliski Jr., B.S.: A Cryptographic Library for the Motorola DSP56000, Advances in Cryptology-Eurpcrypt`90 (LNCS 473), pp.230-244, 1991.

[4] Ploog, H., Timmermann, D.: On Multiple Precision Based Montgomery Multiplication without Precomputation of $N_0' = -N_0^{-1} \, mod \, W$, accepted for presentation ICCD 2000

[5] National Bureau of Standards FIPS Publication 46: DES modes of operation, 1977

[6] Wienke, C.: Hardwareoptimale Implementierung der Montgomery-Multiplikation, University of Rostock, student work (in german), 1999

[7] http://www.rsasecurity.com/rsalabs/des3/