Concurrent Fault Detection in a Hardware Implementation of the $RC5$ Encryption Algorithm

Guido Bertoni[1], Luca Breveglieri[1], Israel Koren[2],
Paolo Maistri[1], Vincenzo Piuri[3]

[1]Department of Electronics and Information Technology
Politecnico di Milano, Milano, ITALY
[2]Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
[3]Department of Information Technologies
Università di Milano, Crema, ITALY

bertoni@elet.polimi.it, brevegli@elet.polimi.it,
koren@ecs.umass.edu, maistri@elet.polimi.it, piuri@dti.unimi.it

Keywords: cryptography, RC5, VLSI architecture, fault detection, error detecting codes.

Abstract

Recent research has shown that fault diagnosis and possibly fault tolerance are important features when implementing cryptographic algorithms by means of hardware devices. In fact, some security attack procedures are based on the injection of faults. At the same time, hardware implementations of cryptographic algorithms, i.e. crypto-processors, are becoming widespread. There is however, only very limited research on implementing fault diagnosis and tolerance in crypto-algorithms. In this paper fault diagnosis is studied for the RC5 crypto-algorithm, a recently proposed block-cipher algorithm that is suited for both software and hardware implementations. RC5 is based on a mix of arithmetic and logic operations, and is therefore a challenge for fault diagnosis. This paper studies fault propagation in RC5, and proposes and evaluates the cost/performance tradeoffs of several error detecting codes for RC5. Costs are estimated in terms of hardware overhead, and performances in terms of fault coverage. Our most important conclusion is that, despite its non-uniform nature, RC5 can be efficiently protected by using low-cost error detecting codes.

1: Introduction

Fault detection and fault tolerance are key issues when implementing cryptographic algorithms. Errors must be avoided as they can heavily corrupt data being transmitted [Bon01] and corrupted data can not be decrypted into meaningful information [Ber02]. The increasing complexity of the most recent cipher algorithms makes this issue more critical than ever.

Furthermore, faults may be intentionally injected by a malicious attacker in order to gather some private information, like the secret key [Bao97]. In [Fer00] a technique for providing fault detection is described: a fixed sequence is attached to each plain text block to be encrypted; the presence of the fixed sequence is checked after decrypting. This technique is quite simple, but highly redundant. Several other papers have already explored fault detection in the AES cipher with various fault models [Ber03, Ber02, Kar01].

The above mentioned papers have all focused on the AES algorithm. In this paper, we extend the study of fault detection to the $RC5$ cipher [Riv95]. This is not, however, a straightforward extension. Unlike AES which consists of only data-independent logical and modular arithmetic operations, the $RC5$ encryption uses data-dependent logical operations (e.g., data-dependent rotation), modular arithmetic operations and conventional arithmetic operations (e.g., two's complement addition). Due to the hybrid nature of the $RC5$ cipher, it is not obvious whether the same class of Error Detecting Codes (EDCs) which proved suitable for AES [Ber03] may be used for $RC5$. We explore in this paper several possible candidate EDCs for $RC5$ including parity codes (which better suit logical and modular operations) and residue codes (which suit conventional arithmetic operations). We estimate through simulation the coverage provided by such codes with different levels of redundancy [Pet72] and carry out a preliminary analysis of the hardware costs of these EDCs.

The paper is organized as follows: in Section 2, a brief description of the $RC5$ algorithm is presented, focusing on the unique characteristics of this algorithm distinguishing it from AES. In Section 3 we analyze the error propagation of single and multiple faults; a study which is helpful for understanding fault coverage. Section 4 presents the candidate error detecting codes – parity and residue codes – and shows how the operations used in $RC5$ affect those EDCs. Section 5 compares the performance of the codes in terms of their fault detection capabilities. Finally, Section 6 summarizes and compares the hardware costs of the proposed solutions.

2: The $RC5$ algorithm

The $RC5$ cipher, proposed by R.L. Rivest in 1995, is a symmetric block cipher system [Riv95]. It was originally developed for a software implementation, but fits hardware implementations as well. The system is parametric, as it can be customized to different security levels and configured to match the word size of the host machine (in the case of software implementation). The main parameter of the algorithm is w, the word size (allowable values of w are 16, 32 and 64). $RC5$ encrypts (and decrypts) an input plain text consisting of two words of size w.

The cipher defines three procedures: Encryption, Decryption (which form the data-path) and Key Schedule (the control-path), which all have an iterative structure. The basic block of Encryption is called round (see Section 2.1) and it is repeated r times (clearly, the parameter r is greater than 0). The Decryption process has a similar structure.

The secret key is composed of $b \geq 1$ bytes, which are expanded into $2(r+1)$ words of size w, according to the Key Schedule algorithm. Each pair of words in the expanded key is called the round key and is added to the plain text being encrypted during the corresponding round. The most commonly used set of parameters is $w = 32$, $r = 16$ $b = 16$. This configuration is denoted by RC5-32/16/16 and will be used in the rest of the paper.

2.1: The data-path

The pseudo-code of the encryption data-path of $RC5$ is shown below. Both A and B are 32-bit registers, each containing an unsigned integer. The array S contains the key material which is obtained using the key expansion procedure. The $RC5$ Encryption algorithm is:

```
// A, B are two 32-bit registers, containing the 64-bit plaintext block}
// S is a 32*r array of words, containing the key material (r >= 1)
A = A + S[0];    B = B + S[1];
for i = 1 to r do
{ A = ((A XOR B) << B) + S[2 * i];    B = (B XOR A) << A) + S[2 * i + 1]; }
// A and B contain the 64-bit encrypted block
```

The 64-bit plaintext block is initially loaded into the register pair A, B. Then, the above algorithm is executed and the encrypted block is found in A, B at the end of execution. The body of the loop represents a single round. The key material is progressively used - two words of the array S at each round.

The decryption of a block is executed iterating an inverse round, which is derived from the encryption round using the inverse of each single transformation applied in reverse order.

2.2: The control-path

The user's secret key is expanded to fill the array S. The key expansion procedure consists of the same basic operations as those used in Encryption and Decryption. The exact details of this Key Schedule procedure are omitted for brevity; the reader can refer to [Riv95].

3: Error analysis

In this section we study the behavior of $RC5$ with respect to the diffusion of errors due to faults occurring during the computation. We first consider the case of a single faulty bit. The behavior of both Encryption and Decryption is analyzed by injecting faults at the beginning of a randomly chosen round and counting the number of erroneous bits in the following rounds and at the end of the complete operation. The results of a large number of such simulations are then analyzed through the mean, the maximum and the minimum values of the number of erroneous bits. In these experiments we used 20 different keys, for each key 20 different plain text blocks; and for each plain text block a single faulty bit was injected at each bit position at the beginning of each round. Each simulation run was carried out for 16 rounds. In total, there were $20 \times 20 \times 16 \times 64 = 409,600$ simulation runs.

The mean number of bit errors quickly converges to a value ≈ 32, which means that a single bit error quickly diffuses uniformly over the entire encrypted data block, i.e., after a few rounds each bit has a probability of 0.5 to be erroneous. The number of errors is distributed in the interval 25% to 75% of the total number of bits; clearly, when the single bit error is injected in the very last rounds, the diffusion of bit errors is less apparent.

The diffusion of bit errors in Decryption when injecting the fault in the Encryption showed that uniformity is reached in a number of rounds slightly larger than that for Encryption.

These conclusions do not change when the single-bit error is injected during the internal operations of a round in the data-path. Moreover, the Key Schedule procedure is expected to exhibit the same behavior, since it uses the same basic operations used in the data-path.

Some simulations on multiple faults have also been carried out, from 2 to 20 injected faults. For each case 10 different secret keys, 100 plain text blocks and 10^4 fault sets were used, for a total of 10^7 tests. The average behavior is comparable to that observed for single fault injections. The only difference is the presence of some fault masking effect. In the set of tests, this behavior occurs only with 2 to 5 faulty bits and is very infrequent. Fault masking occurs 704 times (out of 10^7) with 2 faulty bits, 62 times with 3 faulty bits, 8 times with 4 faulty bits and only once with 5 faulty bits. It never occurred with 6 faults or more.

4: Fault detection techniques

Our objective is to develop fault detection techniques independent of the particular hardware implementation chosen. Hence, it is assumed that the $RC5$ implementation is partitioned into three basic hardware modules: Encryption, Decryption and Key Schedule (an Inverse Key Schedule module is a possible enhancement). Since all three modules have in common the same basic operations, only the Encryption module is examined here in detail; the conclusions will hold for the remaining two modules as well. Note that Key Schedule must be run before starting Encryption (or Decryption). The round keys are stored and accessed whenever necessary. Hence, it is possible to assume that the round keys are protected by some technique and that they are error-free.

4.1: The adopted error detecting codes

There are many types of EDCs [Pet72] but we focus here only on two simple ones: parity and residue. The reason is that these types of EDCs fit the logical and arithmetic operations in $RC5$, and more importantly, these are low-cost EDCs. Both types, however, fit well only some of the operations performed and it is not obvious which EDC is more efficient for $RC5$. We decided therefore to evaluate both types, focusing on a relatively small amount of redundancy. For each one of these types of EDCs we present two implementations and we explore the tradeoff between the fault coverage of the considered EDC and its degree of redundancy (see Table 1).

Type of EDC	**Number of redundant bits**	**Degree of redundancy**
Word parity	1	$1/32 \approx 3\%$
Byte parity	4	$4/32 = 12.5\%$
Residue $2^2 - 1 = 3$	2	$2/32 = 6.25\%$
Residue $2^4 - 1 = 15$	4	$4/32 = 12.5\%$

Table 1. *Proposed error detecting codes – 32-bit words.*

Word parity is comparable to the radix-3 residue code[1], since they exhibit almost the same

[1] Actually it requires two redundant bits, while word parity needs only one, but a single bit residue code

amount of redundancy and both detect all single-bit errors. Byte parity is comparable with the radix-15 residue code: they use four redundant bits, and can detect most two-bit errors.

To implement an EDC in $RC5$ it is necessary to associate check bits with the plain text block to be encrypted, to equip all the internal operations with EDC prediction circuits, to include circuits which compute the EDC initially and recompute it at each checkpoint, and to schedule some checkpoints where the generated and predicted check bits are compared. For simplicity, we choose to schedule a single checkpoint at the end of the last round. In the next sections we define the EDCs in detail and show how to compute and predict the check bits for the various internal operations.

4.2: Structures of the coding schemes

4.2.1: Word parity

Definition. The word parity bit $p(A)$ of A is obtained by XORing all of its 32 bits, i.e.,

$$p(A) = p(a_{31} \dots a_0) = \bigoplus_{i=31}^{0} a_i = a_{31} \oplus \cdots \oplus a_0 \tag{1}$$

Prediction for natural addition. It is well-known (see [Nic99] for a recent review) that the parity of the sum of two natural integers can be obtained by XORing the parities of both summands and of all carries propagated between any two adjacent bits, plus the possible carry-in into the least significant position. Hence:

$$p(A+B) = p(A) \oplus p(B) \oplus C_{in} \oplus \bigoplus_{i=30}^{0} C_{out}^{(i)} \tag{2}$$

where $C_{out}^{(i)}$ is the internal carry from the i^{th} bit to the $(i+1)^{th}$ bit in the addition of A and B ($C_{out}^{(31)}$ is the carry out of the addition, which need not be considered).

Prediction in modulo 2 addition. The parity of the sum is the XOR of the parities of the summands, i.e., $p(A \oplus B) = p(A) \oplus p(B)$.

Prediction in left rotation by $k \geq 0$ positions. Clearly the parity of A is left unaltered, i.e., $p(A <<< k) = p(A)$.

4.2.2: Byte parity

Definition. The parity bit $p_h(A)$ of each of the four bytes of A ($h = 0, 1, 2, 3$) is obtained by XORing the corresponding groups of 8 bits. Hence:

$$p_h(A) = p(a_{8h+7} \dots a_{8h}) = \bigoplus_{i=7}^{0} a_{8h+i} = a_{8h+7} \oplus \cdots \oplus a_{8h} \tag{3}$$

for $h = 0, 1, 2, 3$ where the byte indexed by $h = 0$ is the least significant one.

Prediction in natural addition. The same rule as for a complete word applies here, replicated for each of the four bytes. Hence:

$$p_h(A+B) = p_h(A) \oplus p_h(B) \oplus C_{in}(h) \oplus \bigoplus_{i=6}^{0} C_{out}^{(i)}(h) \tag{4}$$

---

is not feasible.

where $C_{in}(h)$ is the carry out of the addition of the $(h-1)^{th}$ bytes of A and B, respectively, for $h = 1, 2, 3$ ($C_{in}(0) = C_{in}$), and $C_{out}^{(i)}(h)$ is the i^{th} internal carry propagated in the addition of the h^{th} bytes of A and B, respectively, for $h = 0, 1, 2, 3$.

Prediction in modulo 2 addition. The same rule as for a complete word is used, but replicated for each of the four bytes. Thus $p_h(A \oplus B) = p_h(A) \oplus p_h(B)$ for $h = 0, 1, 2, 3$

Prediction in left rotation by $k \geq 0$ positions. This operation is more complex than for a complete word. First note that $p_h(A << k) = p_{h-(k \div 8) \mod 4}(A << (k \mod 8))$ for $h = 0, 1, 2, 3$. Hence, it suffices to consider only the cases $p_h(A << n)$ for $0 \leq n \leq 7$.

Second, two cases can be identified, depending on the size of n, as follows:

$$p_h(A << n) = \bigoplus_{i=7}^{0} a_{8h-n+i \mod 32} = \tag{5}$$

$$= \begin{cases} p_h(A) \oplus \bigoplus_{j=7}^{8-n} a_{8h+j \mod 32} \oplus \bigoplus_{j=7}^{8-n} a_{8(h-1)+j \mod 32} & \text{if } n \in [0,3] \quad (5a) \\ p_{h-1 \mod 4}(A >> (8-n)) & \text{if } n \in [4,7] \quad (5b) \end{cases}$$

The right rotation in case (5b) is dealt with as left rotation. To justify the former case (5a), observe for instance that if $m = 1$ and $n = 2$, we obtain $p_1(A << 2) = p((a_{15} \ldots a_8) << 2) = p(a_{13} \ldots a_6) = p(a_{15} \ldots a_8) \oplus a_{15} \oplus a_{14} \oplus a_7 \oplus a_6$; while for the latter case (5b), if $m = 0$ and $n = 5$, we obtain $p_0(A << 5) = p((a_7 \ldots a_0) << 5) = p(a_2 \ldots a_0 a_{31} \ldots a_{27}) = p((a_{31} \ldots a_{24}) >> 3) = p_3(A >> 3)$ as shown in Figure 1. Note how, splitting into two cases, the maximum number of correction terms to be added is at most 8.

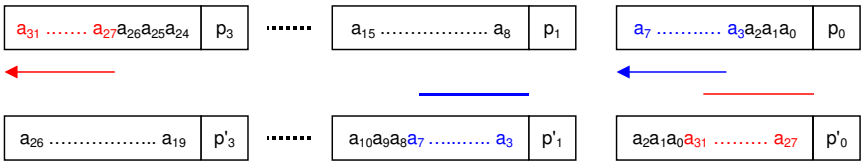

Figure 1. *Long left rotation is replaced by right rotation.*

4.2.3: Residue code

Definition. The residue $r_s(A)$ of A is obtained by taking the modulo $(2^s - 1)$ of A, i.e.,

$$r_s(A) = r_s(a_{31} \ldots a_0) = A \mod (2^s - 1) \tag{6}$$

As indicated in Table 1 we focus on the residue codes for the moduli 3 and 15 which correspond to $s = 2$ and $s = 4$, respectively, in equation (6). In practice, the rules for performing arithmetic operations with the above defined residue are those of one's complement arithmetic, as taking modulo $2^s - 1$ is equivalent to stating that $2^s - 1 = 0$ (e.g., [Kor02]). Since this implies $2^s = 1$, equation (6) can be rewritten as: $r_s(A) = \sum_{i=31}^{0} a_i 2^{i \mod s} \mod (2^s - 1)$.

Prediction in natural addition. The residue of the sum is the sum modulo $(2^s - 1)$ of the residues of the summands, minus a correction term. In detail:

$$r_s(A + B) = r_s(A) + r_s(B) - c_s(A, B) \mod (2^s - 1) \tag{7}$$

for $s = 2, 4$. The correction term $c_s(A, B)$ $(s = 2, 4)$ takes care of the possible carry out when adding A and B as words of 32 bits. In fact, the arithmetic weight of the carry out is 2^{32}, and observing that the chosen values of s divide 32, it follows that $2^{32} \mod (2^s - 1) = (2^s)^{\frac{32}{s}} \mod (2^s - 1) = 1^{\frac{32}{s}} = 1$. Thus,

$$c_s(A, B) = \text{carry out of the addition } A + B \tag{8}$$

Note that when $s = 2$, $-1 = 2 \mod 3$ and hence the correction term is equivalent to $+2c_2(A, B)$.

Prediction in modulo 2 addition. This case is more complex than natural addition. The expression for predicting the residue of the result is:

$$r_s(A \oplus B) = r_s(A) + r_s(B) - 2r_s(A \wedge B) \mod (2^s - 1) \tag{9}$$

for $s = 2, 4$. To justify equation (9), observe that when $a_i \wedge b_i = 1$, (i.e., when two bits of value 1 are XORed), the contributions of these two bits disappears from the result $A \oplus B$, since $a_i \oplus b_i = 0$. The required correction is the negative term $-2r_s(A \wedge B) \mod (2^s - 1)$ in (9). When $s = 2$, $-2 = 1 \mod 3$, and the correction term simplifies to $+r_2(A \wedge B) \mod (2^s - 1)$.

Prediction in left rotation by $k \geq 0$ positions. Denote by $\underset{n}{<<}$ the left rotation of a word of $n \geq 1$ bits. Then,

$$r_s\left(A \underset{32}{<<} k\right) = r_s\left(\left(\sum_{i=31}^{0} a_i 2^i\right) \underset{32}{<<} k\right) = r_s\left(\sum_{i=31}^{0} a_i 2^{i+k \mod 32}\right) =$$

$$= \sum_{i=31}^{0} a_i 2^{(i+k \mod 32) \mod s} \mod (2^s - 1) =$$

$$= \sum_{i=31}^{0} a_i 2^{i+k \mod s} \mod (2^s - 1) = r_s(A) \underset{s}{<<} (k \mod s) \qquad \text{for } s = 2, 4 \tag{10}$$

The simplification $(i + k \mod 32) \mod s = i + k \mod s$ is possible because the chosen values of s divide 32. In practice, the residue of $A \underset{32}{<<} k$ is obtained by rotating by $k \mod s$ positions the original residue of A, which is a sequence of s bits.

4.3: Additional considerations

The EDCs presented in Section 4.2 are relatively simple to apply to $RC5$. In particular, one can make the non-obvious observation that the logical codes (word and byte parity) and the arithmetic codes (residue) can be applied in a simple way to the arithmetic and logical parts of the Encryption procedure, respectively.

The above EDCs and the related prediction techniques can be easily adapted to Decryption and Key Schedule, since both use the same basic internal operations as Encryption. In particular, Decryption uses [Riv95]: modulo 2 addition (as in Encryption); right rotation (the related prediction formulas can be easily derived from those for left rotation); and natural subtraction, i.e., $-A = 2^{32} - A$. The prediction formulas for the latter operation can be derived from those for natural addition, introducing a correction term. The situation is very similar for Key Schedule [Riv95].

5: Fault coverage of the proposed codes

In this section we describe the results of the simulation tests that were performed to evaluate the performance (in terms of fault coverage) of the four EDCs. Fault injection was done following the assumptions stated in Section 3.

The results of the single fault injection tests were satisfactory. Each possible bit position was tested, resulting in $64 \times 17 = 1088$ test cases for each pair of input and key. All the single bit faults were detected by all four EDCs. A similar exhaustive testing was then performed injecting a pair of faults, which means that $\binom{64\times17}{2} = 591328$ test cases were checked for each pair of input and key. Subsequently, three or more faults were injected using random generation of fault vectors, since the search space would have been intractable. The results of these experiments are depicted in Figure 2. As expected, the fault coverage increases with more redundant bits, but there are more interesting conclusions that can be drawn:

1. The use of a single parity bit per word provides a very low protection against multiple faults since about 25% of the test cases were not detected; the residue code with radix-3 performs twice as well but it also requires twice the redundancy;
2. The parity and residue codes with four check bits per word provide almost the same fault coverage except when injecting two faults – in this case, the residue code is better;
3. There is a negligible amount of "false positives" when using parity codes – faults canceling each other in the output, but leading to an erroneous predicted parity. However, this is very unlikely, as it occurred very rarely with 2 or 3 injected faults.

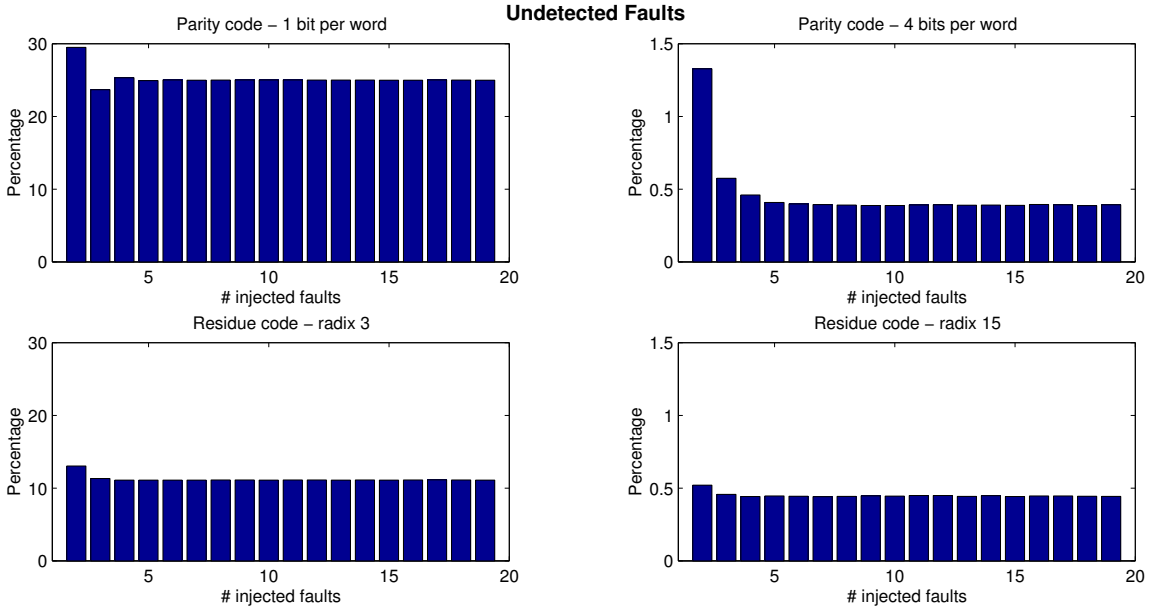

Figure 2. *Frequency of undetected faults, with random test selection.*

All the analyzed EDCs reach quickly an asymptotic behavior as the number of injected faults increases. This is consistent with the fact that errors distribute uniformly after very

few rounds (see Section 3). The byte parity and radix-15 residue codes exhibit the same asymptotic behavior, which can be related to them having the same degree of redundancy.

6: Cost of the proposed codes

We present here a preliminary and approximated evaluation of the hardware costs of the considered EDCs. In all cases, it is assumed that the EDCs are implemented using circuits immediately derived from the various equations shown in Section 5.

In particular, we assume that all circuits are of the combinatorial type. Parity can be computed and predicted by using trees of two-input XOR gates. Residue can be computed and predicted by using trees of full adders (FA) (connected in a carry around structure). Since the prediction of parity sometimes requires the knowledge of the internal carries propagated during integer addition, we assume that $RC5$ is implemented using only adder types making such carries available.

Obviously it is necessary to associate with the plain text block the suitable number of redundant bits, depending on the adopted EDC. This requires to enlarge the registers A and B storing the encrypted pair of words. Table 2 shows the hardware costs of the computation, prediction and check circuits for all four EDCs considered in this paper.

Cost Element	**Type of EDC**			
	Parity		**Residue**	
	Word	Byte	Radix-3	Radix-15
Prediction for				
- Integer Addition	33 XOR	36 XOR	4 FA	8 FA
- Mod. 2 Addition	1 XOR	4 XOR	4 FA, 16 AND	8 FA, 16 AND
- Left Rotation	none	MUX, 32 XOR	MUX	MUX
Other Costs				
- Initial Code Comp.	31 XOR	28 XOR	30 FA	28 FA
- Final Code Comp.	31 XOR	28 XOR	30 FA	28 FA
- Final Code Check	1 XOR	4 XOR	2 XOR	4 XOR
Total Costs	97 XOR	132 XOR, MUX	68 FA, 16 AND, 2 XOR, MUX	72 FA, 16 AND, 4 XOR, MUX

Table 2. *Hardware costs of the various types of EDCs - word of* 32 *bits.*

Table 2 allows us to draw some preliminary conclusions. A Full-Adder (FA) is considerably larger than a XOR gate, hence the residue codes appear to be more expensive than the parity codes. Byte parity and radix-15 residue have the same asymptotic coverage, but the latter appears to be more expensive than the former (though the multiplexing structure for byte parity is relatively complex and deserves a more careful evaluation).

7: Conclusions

A detailed analysis of the performance of various simple and low-cost error detecting codes for integrating built-in fault detection in a hardware implementation of the $RC5$ crypto-

graphic algorithm has been carried out. Error diffusion in $RC5$ has been characterized, techniques to implement the chosen EDC types have been detailed, and the achieved fault coverages have been estimated by simulations.

Some of the presented codes appear to provide an excellent coverage for low-order faults, and a good asymptotic behavior. This leads to the non-obvious conclusion that $RC5$ can be efficiently protected against faults by using such simple techniques as EDCs, despite its non-homogeneous structure.

Further research may include a careful estimation of the hardware overhead of the considered EDCs using VHDL implementations, and possibly the introduction of fault tolerance.

Acknowledgment: The work of Israel Koren has been supported in part by DARPA/AFRL NEST program under contract number F33615-02-C-4031.

References

[Ber03] G. Bertoni, L. Breveglieri, I. Koren. P. Maistri, V. Piuri, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," to appear, *IEEE Transactions on Computers*, Special Issue on Cryptographic Hardware and Embedded Software, 2003.

[Ber02] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "A Parity Code Based Concurrent Fault Detection for Implementations of the Advanced Encryption Standard," *Proc. of the 2002 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 51-59, November 2002.

[Bon01] D. Boneh, R. DeMillo, R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, pp. 101-119, 2001.

[Kar01] R. Karri, W. Kaijie, P. Mishra, K. Yongkook, "Fault-based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture," *Proc. of the 2001 Defect and Fault Tolerance in VLSI Systems*, pp. 418-426, 2001.

[Fer00] S. Fernández-Gòmez, J. Rodríguez-Andina, E. Mandado, "Concurrent Error Detection in Block Ciphers," *Proc. of the 2000 International Test Conference*, ITC '00, pp. 979-984, 2000.

[Nic99] M. Nicolaidis, R. Duarte, "Fault-Secure parity Prediction Booth Multipliers," *IEEE Design and Test of Computers*, 1999.

[Bao97] F. Bao, R. Deng, Y. Han, A. Jeng, D. Narasimhalu, T. Nagir, "Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults," *The Second Workshop on Secure Protocols, (Pads)*, April, 1997, LNCS, Springer-Verlag, 1997.

[Kor02] I. Koren, *Computer Arithmetic Algorithms*, 2nd edition, A K Peters, Natick, MA, 2002.

[Riv95] R. Rivest, "The $RC5$ Encryption Algorithm," *K. U. Leuven Workshop on Cryptographic Algorithms*, Springer-Verlag, 1995.

[Pet72] W. Peterson, E. Weldon, *Error-Correcting Codes*, 2^{nd} ed., The MIT Press, Cambridge, MA, U.S.A., 1972.