Detecting Faults in Integer and Finite Field Arithmetic Operations for Cryptography

L. Breveglieri[*], I. Koren[†] and P. Maistri

Abstract

Fault detection is gaining in importance since fault attacks may compromise even recently developed cryptosystems. In this work, we analyze the different operations used by various symmetric ciphers and propose possible detection codes and frequency of checking. Several examples (i.e., AES, RC5 and DES) are presented to illustrate our analysis.

1 Introduction

Recently, schemes for detecting faults in hardware implementations of several symmetric key encryption algorithms have been developed. The motivation behind the increased interest in such detection schemes is based on two important observations. First, ciphered communication is very sensitive to errors in the input data or faults occurring during the computation, due to the strong non-linearity of the encryption functions. Analysis of the effect of faults occurring during the encryption process, described in [3] for the Advanced Encryption Standard (AES) algorithm and in [4] for RC5, has shown that even a single bit error leads, after only a few rounds of the algorithm, to a completely corrupted result.

The second reason for the increased importance of fault detection, besides the data integrity issue, is the observation that attacks based on fault injection are feasible [6]. The authors of [6] showed that a cryptographic device computing DES could be compromised by injecting a fault during the computation. Depending on the cipher algorithm employed, useful data can be extracted by analyzing the resulting erroneous output. This approach proved to be successful, and was later applied to other more recent algorithms, such as AES [7]. It is worthwhile to note that fault injection attacks are not limited to symmetric block ciphers, although these received the most attention in recent publications. In [2], fault injection attacks against RSA-capable smart cards were studied; in [8], an erroneous RSA signature can lead to an easier factorization of the modulus, thus breaking the cryptosystem.

Some preliminary studies of fault detection schemes have already been performed. In [11], Karri et al. have proposed to use the existing hardware for an immediate decryption of the cipher text. They rely on the fact that the unit responsible for decryption is normally not used during the encryption process and is often independent of the encryption datapath since it involves the inverse transformations. In US patent [10] the embedding of error detection capabilities in DES was proposed: additional bits are generated from the input and provided to the ciphering device in parallel with the plain text. These bits, acting as check bits for the error detecting code, are propagated and updated during the encryption process. Since the cipher is mainly based on substitution tables, these tables are extended in order to also include the check bits. Checking for inconsistencies at the end of the process may thus reveal a possible error in the computation.

A different approach is needed when the operations are more complex than substitutions. Some research has been performed in this direction in [3, 4, 12]. In [3], an error detection code was proposed for the AES cipher. The proposed code is based on the well-known parity code and uses one parity bit for each byte of the 128-bit-long input. In [4], a similar approach is applied to RC5 [15].

[*]L. Breveglieri and P. Maistri are with the Department of Electronics and Information Technology, Politecnico di Milano, Milano, Italy. E-mails: {brevegli,maistri}@elet.polimi.it

[†]I. Koren is with the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst, MA, USA. E-mail: koren@ecs.umass.edu

The objective of this paper is to analyze the different types of basic arithmetic operations employed by various encryption algorithms and list the different fault detection schemes that can be used for these operations. Based on such a classification we can recommend certain fault detection techniques for the analyzed ciphers. Finally, we discuss the required frequency of error checking, i.e., we indicate whether for a given cipher one can wait until the encryption is complete and only then check for errors, or check for errors at a higher frequency (e.g., every round) since the error indication may disappear (due to error masking).

2 Symmetric ciphers

For the purpose of our study we need to first identify the operations that are used by the various ciphers. We carry out this task in this section, and make some preliminary observations regarding fault detection in dedicated VLSI implementations of ciphers. We consider for this purpose two principal approaches to fault detection, namely, duplication (the brute force solution) and the use of an error detecting code. The latter may prove to be more or less efficient than duplication depending on the structure of the selected code.

The list of the symmetric ciphers considered in this paper (see Table 1) is far from being exhaustive; however, it includes all the finalists of the last AES round [1, 9, 14, 16], together with some other previously proposed algorithms like DES [14], and RC5 [15]. Software implementations for all these ciphers are available and all have been used in practice to some extent. However, DES and more recently Rijndael (AES) are more commonly used and dedicated VLSI devices are available for computing them. Still, most of the considered ciphers are well suited for VLSI implementations: for instance RC5 and RC6, which, due to their extreme simplicity, would yield very low cost implementations.

Usually symmetric ciphers have an iterative structure. The encryption process of a data block consists of repeating a number of identical rounds (or two or more alternating round types). Each round may consist of a series of internal transformations, and uses a round key, derived from the secret key. Having an iterative structure greatly simplifies the design and implementation of fault detection mechanisms.

The operations may be applied to the whole data block or part of it and are:

- Bit-wise XOR (Exclusive OR); this is a modular arithmetic (modulo 2) operation.
- Bit-wise AND and OR; these are logical operations.
- Modular addition, subtraction and multiplication of integers; the modulus depends on the considered cipher, but is usually 2^w or $2^w + 1$ (most often $w = 16$ or 32).
- Expansion, meaning that the data block is expanded to a larger number of bits.
- S-Box, or Substitution-Box. It is included in many different ciphers, and consists of a replacement of bytes or words by means of a look-up table. The way it is defined varies with the cipher, and there does not seem to exist any general rule.
- Rotation and shift of the bytes or of the words of the data block.
- Permutation of the bits, bytes or words of the data block.
- Polynomial modular multiplication of the bytes or the words of the data block. Usually one of the two factors is fixed, so that multiplication actually reduces to scaling.

All the above operations, considered in isolation, admit specific error detection codes (EDCs), and some are so simple and inexpensive as to allow duplication, yet there does not seem to exist an EDC which is inherently optimized for all of them. The designers of the various symmetric ciphers have provided some qualitative explanations of the reasons for preferring certain basic operations in the internal transformations. These reasons are based on the need for achieving diffusion (each bit of the input should affect every bit of the output) and confusion (all the regularities of the input are uniformed in the output) while processing the input data block.

The presence of explicitly non-linear operations makes the design of an error detecting code more difficult and the code may become inefficient. The mixture of different and incompatible algebraic structures may also pose a problem: a code which is efficient for one structure may be very inefficient when applied to the other. This may force the insertion of a checkpoint and the generation of new check bits for a different code.

The above mentioned criteria are however qualitative, and still leave many degrees of freedom. We stress that, as far as we know, there exist no specific and well-defined criteria for designing reliable and fault-resistant symmetric cyphers. Moreover, none of the ciphers listed in Table 1 has been implemented in a way that specifically addresses these issues.

We next make some detailed observations regarding Table 1. First, there is only one operation, namely XOR, which is used in all the ciphers analyzed. Note also that there is only a single cipher, i.e., Camellia, which makes use of purely logical operations (AND and OR) (which, incidentally, are well-known to be hard for error detecting codes). The same is true for shift, which is used only by Serpent; expansion is used only by DES. Polynomial multiplication (or scaling) is common to only two ciphers: Rijndael (AES) and Twofish. All the remaining operations are common to three or more ciphers.

Some operations in Table 1 are invertible, while others are not (e.g., AND, OR, shift, etc). Every cipher is invertible, since decryption must be possible. The presence of non-invertible operations is compensated by the fact that the input operands are preserved in some way during the process. These operands are also forwarded in some way, so that during decryption they can be reconstructed. It is well-known that the presence of multiple data flows is a challenge for fault detection.

In what follows, the analysis of the fault detection issues is restricted to the Encryption data-path of the ciphers, excluding both the Key Schedule and the Decryption parts. This is done both for simplicity of analysis and because Key Schedule is usually built around the same operations used in Encryption. Table 1 shows the sizes of the data processed by the internal operations of the ciphers, but restricts attention to the Encryption data path only, for the above mentioned reasons. For the S-Box, both the input and the output size are listed (since they may differ).

Some details regarding Table 1 are necessary for the rest of the paper. The operations $(+)$ and $(-)$ are integer addition and subtraction, respectively, modulo 2^w with $w = 16$ or 32. The operation $(\times)$ is integer multiplication modulo 2^{32} or $2^{16}+1$ (the latter modulus is used by IDEA [14]). Expansion is used only by DES, and consists of transforming 4-bit nibbles into sequences of 6 bits. S-Box is a substitution of bit sequences (and its definition depends on the cipher) with the most frequent case being a byte substitution. Since S-Boxes must be invertible, they are one-to-one functions. Rotation and shift are simple operations and the number of bit positions may be either constant or data dependent. Permutation consists of the exchange of bits, bytes or words. Polynomial multiplication is defined over the finite field $GF(2^8)$ with one factor always a constant, i.e., the operation reduces to scaling.

3 Operations and error detection codes

This section provides a brief overview of the fault detection techniques which are used in this paper and discusses their application to the selected ciphers. We focus on the basic multiple transient fault model, which is appropriate for algorithms working on relatively large data blocks (64 to 128 or more bits). Although our analysis is based on the hypothesis of multiple bit errors, we still consider and analyze first the single bit error, which allows us to describe the model more precisely.

Two types of error detection techniques are available: duplication followed by comparison of the results, and the use of error detecting codes (EDCs). Clearly, duplication can be applied to every cipher and would achieve a 100% fault coverage. It is however a brute force solution to error detection, and has a high hardware overhead.

Error detecting codes are, in principle, more promising than duplication: they may achieve a relatively high coverage, at least for low order errors, with a relatively low hardware overhead. Moreover, EDC's could be applied to the entire data block, or to parts thereof (bytes, words), thus allowing several variants. We therefore mainly focus in the next sections on employment of EDC's. The basic ones are: arithmetic residue codes with the modulus 3, 7 and 15; and parity codes. Residue codes are most suited for modular arithmetic operations, while parity codes are appropriate for logical and polynomial ones.

Remember that the use of these EDC's implies the need of: a code generator circuit (to be used initially and ahead of every checkpoint), a set of code prediction circuits (one for each internal operation), and a comparator for checking the real check bits against the predicted ones at each checkpoint. The scheduling of the checkpoints must also be planned, and it depends on the tradeoff between the desired fault coverage and the acceptable hardware and performance overhead. Since all ciphers are iterative, scheduling the checkpoints basically means deciding on the checkpoint frequency in the round flow, and whether the checkpoint should be executed between two rounds or in the middle of the round.

All the above listed EDC's can be applied at different levels of granularity. Table 1 suggests that the right levels to select from are byte level (8 bits) and word level (16 or 32 bits), since most internal operations work on data of such sizes. Applying an EDC at the level of the entire data block does not seem to be a good choice since the data block is large and is always fragmented into shorter bit sequences for processing: a global code would require a large overhead for prediction and will provide a low fault coverage. We will therefore consider residue and parity codes at the byte and word level.

3.1 Feasibility of the EDC techniques

Table 2 shows the estimated cost of the various EDC techniques listed above, when applied to the operations of the ciphers, listed in Table 1. We distinguish between EDC's which have an acceptable cost (a "yes" entry in the table) and expensive EDC's (an "exp." entry). An EDC is considered expensive when its implementation (generation, prediction and comparison of the check bits) requires an overhead comparable to that of duplication, which is 100% plus the overhead of the comparator of the two results. Some special cases in Table 2 are explained below, in Sections 3.1.1 and 3.1.2.

3.1.1 Codes

We next provide explanations on the entries of Table 2. The generation and prediction techniques of the various EDC's are omitted for brevity. Residue base 15 is not considered at the byte level since it uses 4 check bits resulting in 50% bit overhead, which is unacceptable for an error detection code; it is considered at the level of words of 16 or 32 bits.

Both parity and residue coding are reasonable for bit-wise XOR, although parity is obviously more suitable. On the other hand, both codes are expensive for bit-wise AND and OR; moreover, AND and OR are so inexpensive themselves, that the prediction of any conceivable EDC will cause an overhead larger than that of duplication.

Expansion is used only by DES: due to its fine-grained and unconventional size of input to the S-Box, any code is quite expensive. S-Box is a non-linear substitution, and hence its treatment is more complex and will be detailed below.

Both parity and residue codes are feasible for integer modular addition and subtraction; still, residue is better suited than parity code, which can be an acceptable solution for a single byte. Residue code is appropriate for integer modular multiplication, while parity is expensive since all the intermediate carries must be considered. In contrast, parity is feasible for polynomial multiplication in $GF\left(2^8\right)$ when one of the two factors is fixed (scaling), while the residue code is expensive.

Both parity and residue codes incur a reasonable overhead for rotation at the byte and word level, with the exception of residue base 7. Shifting is generally expensive because the shifting amount may not be known a priori; however, it is used only in Serpent and only small shift amounts are used, which means a reasonable overhead for parity and residue codes (since shifting by k is a multiplication by 2^k); both parity and residue codes have reasonable overhead for permutations at the byte and word levels, obviously.

3.1.2 S-Box

S-Box is a non-linear substitution which is usually implemented by means of a look-up table. Two kinds of faults are possible: those affecting the contents of the look-up table and those affecting the address decoder. The address is extended by concatenating the check bits of the code. In the entries of the look-up table for valid address codewords, the corresponding correct output codewords are stored (data bits plus check bits), while the remaining entries contain a de-

		Operations - Data-Path of the Encryption part of the Cipher											
	Input				mod n								mod $G(x)$
Ciphers	**Size**	XOR	AND	OR	$+$	$-$	$\times$	Expan.	S-box	Rot.	Shift	Perm.	$\times$
Camellia	128	8, 32, 64	32	32					$8 \rightarrow 8$	32		32, 64	
DES	64	32, 48						$32 \rightarrow 48$	$6 \rightarrow 4$			1	
IDEA	64	16			16		16					16	
MARS	128	32			32	32	32		$8 \rightarrow 32$ $9 \rightarrow 32$	32		8	
RC5	64	32			32					32			
RC6	128	32			32		32			32		8	
Rijndael	128	8							$8 \rightarrow 8$			8	8
Serpent	128	32							$4 \rightarrow 4$	32	32		
Twofish	128	32			32				$8 \rightarrow 8$	32		8, 64	8

Table 1: Symmetric ciphers and the operations they use in the Encryption data-path only, with operand size.

	Per byte			Per word			
Operation	**Parity**	**Res.** 3	**Res.** 7	**Parity**	**Res.** 3	**Res.** 7	**Res..** 15
XOR	yes	yes	yes	yes	yes	yes	yes
AND	more expensive than duplication						
OR	more expensive than duplication						
$+$ mod n	yes	yes	yes	yes	yes	yes	yes
$-$ mod n	yes	yes	yes	yes	yes	yes	yes
$\times$ mod n	exp.	yes	yes	exp.	yes	yes	yes
Expansion	separate treatment (applies only to DES)						
S-Box	yes	exp.	exp.	yes	exp.	exp.	exp.
Rotation	yes	yes	exp.	yes	yes	exp.	yes
Shift	yes	yes	yes	yes	yes	yes	yes
Permutation	yes	yes	yes	yes	yes	yes	yes
$\times$ mod $G(x)$	yes	exp.	exp.	yes	exp.	exp.	exp.

Table 2: Operations and the allowed error detection coding techniques: exp. - very expensive; yes - reasonably feasible.

liberately incorrect codeword (e.g., the data can be all 0 with invalid check bits). This way, the data section of the memory is protected, but not the address decoder.

To protect the latter, an auxiliary and independent memory unit is needed, storing only the check bits of the correct output. The two memories are operated in parallel and the check bits of the codeword provided by the main memory must be compared with the output bits of the auxiliary memory. A mismatch between the two will indicate a fault in the address decoder. When this happens, the system should output a deliberately incorrect codeword, as before.

In both cases, the address is extended by concatenating the check bits. Therefore, only the parity code has an acceptable overhead since it doubles the size of the memory, which has an overhead similar to that of duplication. Residue codes with modulus 3, 7 or 15 would increase the size of the memory by a factor 4, 8 or 16, respectively, which is unacceptable. Note also that the auxiliary memory (for detecting address decoder faults) is much smaller than the main one and its overhead is relatively small.

3.2 Preferred EDC's

Some preliminary conclusions can be drawn regarding the preferable error detection code for each symmetric cipher. Most operations allow simple prediction rules both for parity and residue codes. Few operations are an exception to this rule: integer multiplication, for instance, allows only for residue prediction: this forces to choose residue codes when the cipher employs integer multiplications, such as RC6. On the other hand, expansion in DES and polynomial multiplication (AES and Twofish) are better suited to parity codes, which are hence the suggested choice for those ciphers.

IDEA uses only exclusive ORs, natural additions and modular multiplications. However, the product uses the modulus $(2^{16} + 1)$ so that the parity code is not a reasonable choice, but even residue codes are expensive, since the unusual modulus makes the computation of the corrective term a very complex task.

As stated in Section 3.1.2, the addressing and storage overheads for residue code prediction in S-Boxes suggest that parity code is preferable. However, MARS uses also integer multiplications, which are not suited for parity prediction, hence a compromise must be achieved. When no conflicts exist, parity is still the simplest choice. Other operations have affordable prediction rules for both codes which can hence be reasonable choices, such as the case of RC5. The results are summarized in Table 3.

Cipher	**Suggested Code**
Camellia	Intractable by EDC
DES	Parity
IDEA	Residue, but expensive
MARS	Residue, but expensive
RC5	Parity or Residue
RC6	Residue
Rijndael (AES)	Parity, per byte
Serpent	Parity, per byte
Twofish	Parity, per byte

Table 3: Suggested error detecting codes for protecting various symmetric ciphers.

4 Frequency of checking

In the previous sections various options have been examined for detecting faults in the studied symmetric ciphers. All the detection techniques (i.e., parity and residue, at the byte and word level), are able to detect single transient faults (see for instance [13]). This means that, if error checking is performed at the end of each internal transformation of every round, the coverage of single bit transient faults is 100%.

However, this is a very high checking frequency and has a considerable hardware overhead. Since the ciphers are all iterative and consist of a repetition of a basic round, it might be possible to perform the checking less frequently, for instance once at the end of each round or even only once at the end of the whole sequence of rounds. This allows speeding up the clock rate and reduces the time latency. In a pipelined architecture, reducing the check frequency would allow implementing fewer checkers and thus reducing the hardware overhead as well.

Define the *error signature* as the difference between the real values and the predicted values of the check bits for the data block. Then, the exact error propagation model will depend on the particular cipher studied. We assume that the key scheduling algorithm is fault-free, and the rounds following the one(s) that has (have) been affected by the fault are also fault-free: consequently, they *evolve* the error signature, by spreading or canceling the errors.

To determine how an error signature propagates throughout the cipher, it is necessary to examine how the predicted values of the check bits at the *output* of each transformation contained in the round step depend on the predicted values of these bits at the *input* of the transformation.

In simple cases, the propagation rule of the error signature depends only on the value of the current error signature (and not on the actual values of the code or the datum). This is not always true and must be confirmed in every case, though most EDC's satisfy it. Should this be false, the analysis of the propagation of the error signature would be much more complex, since it would depend on the datum as well.

Thus, the error signature depends only on its previous value and is determined by the single prediction rules. In most cases, the global rule proves to be a linear error propagation model or is reducible to a composition of linear models, thus simplifying considerably the error signature propagation rules.

The error propagation analysis is therefore very dependent on the considered cipher. The analysis for AES is partially outlined in [3], and can serve as a guideline for the remaining ciphers. The checkpoints for AES were scheduled only at the end of the encryption: as shown in [5], the evolution of the error state model can be described by a 16×16 matrix defined over $GF(2)$: this matrix is non-singular and its 8^{th} power is the Identity matrix. Hence, any single fault will propagate (spreading and contracting) to the end and will never be completely canceled. Even the RC5 cipher gives similar results [4]: both residue and parity codes can be propagated through the operations used in RC5, and the propagation model (not described here for brevity) allows the single fault to reach the comparator at the end of the encryption.

DES requires a completely different approach: due to its permutation unit, which acts at the bit level on the whole word, it is not possible to derive a simple prediction rule for the detecting code. Hence, a checkpoint must be scheduled *within* the round, and then the code has to be regenerated. The inner checkpoint makes the check at the round level redundant: this is the approach used in [10], where only inner and final checkpoints are scheduled, but none at the end of the round. IDEA would also require internal checkpoints, due to the prediction overhead for its multiplications. However, since there are 4 checks per round, implementing them would be extremely costly in terms of overhead and latency.

Such frequent checking is also required when the code is larger-grained than the operations. Consider for example, 8-bit S-Boxes and a word parity code; before accessing the look-up table, we need to check the consistency of the code, compute the S-Box and finally recompute the new code, with an additional global overhead.

5 Conclusions

In this paper, some suggestions were given in order to provide fault detection capabilities in recent block ciphers. Some preliminary experiments were run, but the results were not shown for brevity.

Acknowledgment: The work of Israel Koren has been supported in part by DARPA/AFRL NEST program under contract number F33615-02-C-4031. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Projects Agency, AFRL, or the US Government.

References

[1] R. Anderson, E. Biham and L. Knudsen, "*Serpent: A Proposal for the Advanced Encryption Standard*," available from www.cl.cam.ac.uk/ ~rja14/serpent.html, 1999.

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures," http://citeseer.nj.nec.com/525626.html, 2002.

[3] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "Error analysis and detection procedures for a hardware implementation of the Advanced Encryption Standard," *Computers, IEEE Transactions on*, Volume 52, Issue 4, pp. 492-505, 2003.

[4] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "Concurrent Fault Detection in a Hardware Implementation of the *RC*5 Encryption Algorithm," *Proc. of the IEEE Intern. Conf. on ASAP '03*, pp. 410-419, 2003.

[5] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "Detecting and locating faults in VLSI implementations of the AES," *Proc. of 18th IEEE Intern. Symp. on DFT in VLSI Systems, '03* pp. 105 - 113, 2003.

[6] E. Biham, A. Shamir, "Differential Cryptanalysis of the Data Encryption Standard," Springer Verlag, 1993.

[7] J. Blöemer and J.-P. Seifert, "Fault based cryptanalysis of the Advanced Encryption Standard," Cryptology ePrint Archive, Report 2002/075, 2002.

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

[9] C. Burwick, et al., "*MARS - A Candidate Cipher for AES*," NIST AES Proposal, researchweb.watson.ibm.com/security/mars.pdf, 1998.

[10] A. S. Butter, C. Y. Kao, J. P. Kuruts, "DES encryption and decryption unit with error checking," US patent US5432848, July 1995.

[11] R. Karri, K. Wu, P. Mishra, K. Yongkook, "Fault-based side-channel cryptanalysis tolerant Rijndael symmetric block cipher architecture," *Proceedings of the IEEE Intern. Symposium on DFT in VLSI Systems '01*, pages 427-435, 2001.

[12] R. Karri, G. Kuznetsov, M. Goessel, "Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers," *Proceedings of CHES 2003*, Springer-Verlag, pages 113-124, 2003.

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

[14] B. Preneel et al., "NESSIE D20 - NESSIE security report," citeseer.ist.psu.edu/preneel03nessie.html, 2003.

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

[16] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson, "*Twofish: A 128-Bit Block Cipher*," CounterPane Labs, available at http://www.counterpane.com/twofish.pdf, 1998.