An Operation-Centered Approach to Fault Detection in Symmetric Cryptography Ciphers

Luca Breveglieri, Israel Koren, *Fellow*, *IEEE*, and Paolo Maistri

Abstract—One of the most effective ways of attacking a cryptographic device is by deliberate fault injection during computation, which allows retrieving the secret key with a small number of attempts. Several attacks on symmetric and public-key cryptosystems have been described in the literature and some dedicated error-detection techniques have been proposed to foil them. The proposed techniques are ad hoc ones and exploit specific properties of the cryptographic algorithms. In this paper, we propose a general framework for error detection in symmetric ciphers based on an operation-centered approach. We first enumerate the arithmetic and logic operations included in the cipher and analyze the efficacy and hardware complexity of several error-detecting codes for each such operation. We then recommend an error-detecting code for the cipher as a whole based on the operations it employs. We also deal with the trade-off between the frequency of checking for errors and the error coverage. We demonstrate our framework on a representative group of 11 symmetric ciphers. Our conclusions are supported by both analytical proofs and extensive simulation experiments.

Index Terms—Cryptography, symmetric cipher, error-detecting code, parity code, residue code, fault attacks, fault detection.

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 nonlinearity of the encryption functions. The analysis of the effect of faults occurring during the encryption process for the Advanced Encryption Standard (AES) algorithm [6] and for RC5 [7] has shown that even a single-bit error leads, after a few rounds of the algorithm, to a completely corrupted result. The second reason for the increased importance of error detection is the observation that attacks based on fault injection are feasible [9]. The authors in [9] show that a cryptographic device computing the Data Encryption Standard (DES) can be compromised by injecting a fault during the computation. Depending on the cipher employed, useful data can be extracted by analyzing the resulting erroneous output. By detecting the fault, either the output can be blocked (by producing a constant value such as all zeros) or a random output can be generated, misleading the attacker.

- *L. Breveglieri is with the Department of Electronics and Information Technology, Politecnico di Milano, Milano, Italy. E-mail: brevegli@elet.polimi.it.*
- *I. Koren is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003. E-mail: koren@ecs.umass.edu.*
- *P. Maistri is with TIMA Lab., Grenoble, France. E-mail: maistri@elet.polimi.it.*

Manuscript received 5 Aug. 2005; revised 7 Aug. 2006; accepted 16 Oct. 2006; published online 6 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0258-0805.
Digital Object Identifier no. 10.1109/TC.2007.1015.

Techniques for injecting faults into a cryptographic device are readily available and inexpensive; see [5] for a recent and comprehensive review thereof and [24] for the detailed description of an experimental fault injection attack. Such techniques range from the simple exposure of the device to a camera flash, to the injection of an electrical glitch into the power supply, or the use of a laser beam.

The importance of avoiding errors during encryption was first discussed in [11], where it is shown how an erroneous Rivest-Shamir-Adleman (RSA) signature can lead to an easier factorization of the modulus and the breaking of the cryptosystem. This approach was later applied to more recent algorithms such as AES [10], [18], [29]. Fault injection attacks have been developed for other public-key ciphers. In [17], it is shown how an error in some parameters of an Elliptic Curve Cryptosystem may reveal information that could lead to the secret key and, in [4], new fault injection attacks against RSA-capable smart cards were studied.

Hence, fault diagnosis has gained importance in recent research [12] and some preliminary studies of fault detection schemes have already been performed. In [20], use of the existing hardware for an immediate decryption of the ciphertext was proposed, relying on the fact that the decryption unit is normally not used during encryption. The authors propose three different approaches for detecting faults in the encryption process with different impacts in terms of time and complexity. The simplest solution (with the longest detection latency) executes online decryption after the encryption has completed. The other two perform the checks for faults after every single round or after each single operation within the round.

Such an approach is generic and does not exploit any specific property of the encryption algorithm, except the presence of the dual (ciphering-deciphering) components. Hence, it can be applied to virtually any round-based

cipher, which describes most symmetric key cryptosystems. Conceivably, more effective solutions can be devised when targeting a specific cipher. In [16], 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 plaintext. These bits, acting as check bits for the error-detecting code (EDC), are propagated and updated during the encryption process. The cipher's substitution tables are extended in order to include the code check bits. Checking for inconsistencies at the end of the process may thus reveal a possible error in the computation.

Although this approach accomplishes its goal, it suffers from several drawbacks. First, the path that the code must traverse from input to output is interrupted within the round due to the bit permutation, necessitating an additional check within the encryption data path rather than at the end of the round or encryption. This makes the design more complex. Moreover, due to the combination of expansion and substitution boxes (S-boxes), which forbids a simple and affordable prediction rule for the check bits, these must be recomputed using the then available data bits. The code may hence be consistent even if the data is already corrupt. Furthermore, this approach is less useful for modern ciphers, where the substitution tables (or S-boxes) are just one of many operations executed in each round. Applying a table approach to other types of operations (for example, data-dependent rotations or arithmetic operations) would require excessively large tables.

A different approach is needed when the operations are more complex than substitution. In [6], a parity-based error detection code was proposed for the AES cipher using one parity bit for each byte of the 128-bit-long input. The redundancy level was determined based on the round operations (that include *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*, which are byte-oriented logical functions (for a complete description of these operations, see [19] or [27]). The work presented in [6] also shows how the parity code can be propagated through all of the round operations, achieving an impressive fault coverage (fault detection probability): All odd-order faults (that is, an odd number of bit errors has been injected) are detected (the theoretical proof is given in [8]) and the overall fault coverage is over 99 percent. An implementation of the detection scheme was presented in [14], where it was shown that a simple architecture can be augmented with error detection capabilities and acceptable overheads (both in terms of space and time).

In [7], a similar approach is applied to RC5 [31]. The operations included in the RC5 encryption process cover a wider spectrum than those used in AES and include a mix of logical operations (for example, data-dependent rotation and exclusive OR) and arithmetic operations (for example, natural integer addition and subtraction). This mixture of operations makes the selection of a fault detection code nontrivial. The parity code fits logical operations better than arithmetic codes, whereas the opposite is true for a code based on residues [28]. Moreover, RC5 operates on words whose size is machine-dependent. It turns out that the level of redundancy can be tuned to the machine word size or can be set to a finer level (for example, one parity bit per byte). The work presented in [7] shows that, when using such simple codes, a high detection probability is obtained, depending on the level of redundancy.

Karri et al. later proposed in [21] the use of a single parity bit for the whole data block. The coverage is still complete for single-bit errors and the cost is lower due to the lower redundancy. In [36], they developed a concurrent error detection approach for involution ciphers (those where encryption and decryption functions coincide), exploiting the ambivalence of the encryption process.

Different methods of protecting the RSA cryptosystem include [34], where Shamir proposed using a multiplicative masking to foil timing and fault attacks, and [35], where Walter suggested the use of residue codes to protect the modular arithmetic operations in RSA. The error coverage of such codes depends on the value that is chosen for the base modulus of the residue, whereas their cost is comparable to the cost of an extra digit in the operands.

The approaches described in the literature focus only on specific ciphers and attacks. In this paper, we present a general operation-centered approach to error detection. We first list the different types of basic logic and arithmetic operations employed by the various encryption ciphers and the different EDCs that can be used for each such operation. Next, we recommend an EDC for each cipher based on the operations it uses and the hardware complexity of the EDCs that fit them. We then discuss the frequency of error checking, which can be done at different granularities—after every operation, after every round, or only once, at the end of the encryption. The latter is clearly the cheapest, but involves the risk of missing some error indications due to error masking. When determining the checking frequency, we take into account this trade-off between hardware overhead and error coverage, supporting our findings by both analytical and simulation results.

The paper is organized as follows: In Section 2, we introduce the basics of symmetric cryptosystems and describe the spectrum of operations found in a selected list of 11 ciphers on which we concentrate. The fault models that we use and our approaches to fault detection are presented in Section 3. In Section 4, we show which error detection codes are suitable for each operation. In Section 5, we outline our recommendations regarding the most appropriate code to be used for each of the 11 ciphers on our list based on the estimated hardware overhead. The appropriate frequency of checkpoints with the objective of maximizing fault coverage for single-bit faults is discussed in Section 6. The ciphers AES, DES, and the International Data Encryption Algorithm (IDEA) are used as examples and the proof of the complete fault coverage for RC5 is given in Appendix B. The discussion is extended to the coverage of multiple errors in Section 7, based on simulation experiments. Appendix A lists the techniques for check bits prediction for the various operations. Finally, conclusions are presented in Section 8.

TABLE 1
Parameters of 11 Symmetric Ciphers

Cipher	Size of the Atomic Data Block	Input Size	Key Size
Blowfish	32	64	32 to 448
Camellia	8	128	128, 192, 256
CAST-256	32	128	up to 256
DES	8	64	56
IDEA	16	64	128
MARS	32	128	128 to 448
RC5	32	64	up to 2048
RC6	32	128	up to 2048
Rijndael (AES)	8	128	128, 192, 256
Serpent	32	128	up to 256
Twofish	32	128	up to 256

2 Internal Operations in Symmetric Ciphers

2.1 Symmetric Ciphers

We discuss in this paper a list of 11 symmetric ciphers (see Table 1), which is far from being exhaustive. Still, it includes all of the finalists of the last encryption standard competition in addition to some other well-known algorithms, like DES [25], Blowfish [32], CAST-256 [1], and RC5 [31]. All of these ciphers have software implementations and have been used in practice to some extent. DES and, more recently, Rijndael (AES) are more commonly used and have dedicated VLSI devices for their computation. Most of the other ciphers in Table 1 are also well suited for VLSI implementations. RC5 and RC6, for example, would yield very low cost implementations due to their extreme simplicity; specifically, RC5 has recently gained popularity in sensor networks.

Symmetric ciphers usually have an iterative structure. Encrypting a data block consists of repeating a number of identical rounds (or several alternating round types). Each round consists of a series of internal transformations and uses a round key derived from the secret key. Decryption is simply the inverse process of encryption and has a similar structure. All ciphers consist of three main parts: encryption, decryption, and key schedule. The key schedule is the auxiliary algorithm for computing the round keys and has an iterative structure as well. The set of all round keys is called the *key material*. In some cases, an inverse key schedule is implemented as well unless the key material is computed only once and kept stored as long as the secret key is not replaced.

Although most of the ciphers listed in Table 1 accept a 128-bit input (in order to comply with the requirements imposed by the US National Institute of Standards and Technology (NIST)), the older ones usually have smaller inputs or keys (but some can admit up to 256 bits). The width of the input and output and the secret key size are summarized in Table 1 for the 11 symmetric ciphers on which we concentrate.

2.2 Internal Operations

The internal operations used by each cipher differ greatly and the combined list is quite extensive, including almost all obvious elementary logical and arithmetic operations (integer and modular). Table 2 shows the internal operations and the sizes of their operands for the ciphers listed in Table 1. For simplicity, we restrict ourselves to the encryption data path only. The key schedule is usually built around the same operations used in encryption and feasible solutions for encryption will apply to the key schedule with little additional effort. Decryption simply consists of the reverse rounds of encryption, repeatedly applied in reverse order. The operations listed in Table 2 are

- bitwise XOR (exclusive OR)—this is a modular arithmetic (modulo-2) operation,
- bitwise AND and OR—these are logical operations,
- modular addition, subtraction, and multiplication of integers—the modulus varies depending on the cipher, but is usually of the form 2^w or $2^w + 1$,
- expansion—meaning that the data block is expanded to a larger number of bits,
- S-box—this is a peculiar internal transformation of many different ciphers, consisting of replacing bytes or words using a lookup table, with the purpose of introducing nonlinearity into the algorithm,

TABLE 2
Size of Operands in Operations Used in the Encryption (or Decryption) Data Path of Symmetric Ciphers

	Operations in the Data-Path of the Encryption Part of the Cipher											
				mod n								mod $G(x)$
Ciphers	XOR	AND	OR	+	−	×	Expan.	S-box	Rot.	Shift	Perm.	×
Blowfish	32			32				$8 \rightarrow 32$				
Camellia	8, 32, 64	32	32					$8 \rightarrow 8$	32		32, 64	
CAST-256	32			32	32			$8 \rightarrow 32$	32			
DES	32, 48						$32 \rightarrow 48$	$6 \rightarrow 4$			1	
IDEA	16			16		16					16	
MARS	32			32	32	32		$8, 9 \rightarrow 32$	32		8	
RC5	32			32					32			
RC6	32			32		32			32		8	
Rijndael	8							$8 \rightarrow 8$			8	8
Serpent	32							$4 \rightarrow 4$	32	32		
Twofish	32			32				$8 \rightarrow 8$	32		8, 64	8

- rotation and shift of the bytes or words of the data block,
- permutation of the bits, bytes, or words of the data block, and
- polynomial modular multiplication with, usually, one of the operands fixed, reducing it to scaling.

Consider the permutation operation as an example: 1 means that individual bits are exchanged in the data block, 8 that bytes are exchanged, 32 that words are exchanged, and so on. For the S-box, both the input and the output size are listed (since they may differ). The sizes of the data block and secret key are listed in Table 1 and are not repeated here.

Some operations in Table 2 are invertible (for example, addition, subtraction, and rotation), while others are not (for example, AND, OR, and shift). Remember that every cipher must be invertible for decryption to be possible. The presence of noninvertible operations is compensated for by preserving their input operands in some way during the process and forwarding them so that they can be reconstructed during decryption. Such multiple data flows add an extra challenge to fault detection.

The basic operations are chosen based on the need to achieve the following important cipher properties: *diffusion*—each bit of the input affects every bit of the output—and *confusion*—all of the regularities of the input are obscured in the output while processing the input data block. These requirements are achieved by

- Mixing linear and nonlinear operations—linear operations are low cost and helpful for diffusion, whereas nonlinear ones are more expensive but make the cipher harder to invert (thus preventing cryptanalysis).
- Mixing algebraic structures of different types to prevent the existence of a simple mathematical description of the cipher that could be used to attack it.

The presence of explicitly nonlinear operations (for example, in AES) makes error detection more difficult. The mixture of different and incompatible algebraic structures (for example, in RC5) may also pose a problem: A technique that is efficient for one structure may be very inefficient when applied to another.

Some additional details regarding Table 2 can help in understanding the rest of the paper:

- The operators "$+$" and "$-$" are integer addition and subtraction, respectively, modulo 2^w with $w = 16$ or 32.
- The operator "$\times$" is integer multiplication modulo 2^{32} or $2^{16}+1$ (the latter modulus is used by IDEA).
- Expansion is used only by DES, transforming 4-bit nibbles into sequences of 6 bits.
- S-box is a substitution of bit sequences (with the exact definition depending on the cipher), with the most frequent case being a byte substitution. These are one-to-one functions.
- Rotation and shift are simple operations; the number of bit positions may be either constant or data-dependent.
- Permutation means exchanging bits, bytes, or words.
- Polynomial multiplication is defined over the finite field $GF(2^8)$ (that is, it operates on bytes).

The next section describes the model we use to determine suitable error detection techniques for the operations and, consequently, for the ciphers.

3 Fault Model and Detection Techniques

We assume that the basic architecture of each of the ciphers studied here consists of the implementation of one encryption round by means of a dedicated device which is activated sequentially for the required number of rounds. This is the most natural and straightforward way to implement a cipher in VLSI. For high performance applications, pipelined implementations can be considered; the fault detection analysis in the pipelined case will not be much different.

The fault model we focus on is the transient bit error model. This model assumes that each bit can be flipped (0 to 1 or vice versa) with some probability p and that the error goes away after a very short time. This model fits both random errors and security attacks; most benign random hardware faults are short-lived and an attacker is interested in inflicting only a transient failure and not in breaking the device.

As for the number of simultaneous errors, we deal with two cases: One allows at most a single-bit error per data block, whereas the other one allows multiple errors. Again, both cases are practical for either random failures or attacks. Random failures can be either of the single-bit error type or of the correlated type that causes multiple errors in neighboring bits. An attacker may try to flip exactly one bit (gaining more information this way), but, since current attack tools are not very precise [5], [24], either one or several bits will actually be flipped.

For the single-bit-error case, we will present analytical results, whereas, for the multiple-error case, we will rely on simulations. In our simulations, we considered transient faults with up to 20 bit errors for a single plaintext, occurring at random locations and at random times.

We consider two types of error detection techniques: 1) *duplication* followed by comparison of the results (the brute force solution) and 2) the use of EDCs.

Clearly, duplication can be applied to every cipher and will achieve a 100 percent fault coverage. This is true for both the single and the multiple-bit errors and for both random errors and fault attacks as long as the fault is transient. It is highly unlikely that two random transient faults in the duplicated units will have the same effect and produce the same wrong result. As for attacks, an attacker will often be able to inject a fault but not to precisely control its location. In both cases, the two encrypted results will differ and the fault will be detected after output comparison.

EDCs may prove to be either more or less efficient than duplication, depending on the structure of the selected code. Although simple ciphers like RC5 and RC6 may possibly be completely duplicated, for other more complex ciphers, EDCs may achieve a relatively high coverage with a lower hardware overhead. EDCs are especially attractive since the size of the data block is usually high and most codes add to the data block only a fixed and limited number

TABLE 3
Normalized Hardware Complexity (with Respect to XOR) of the Various Operations and the Different EDCs

Operation	per Byte				per Word				
	Size / XOR	Parity	Res. 3	Res. 7	Size / XOR	Parity	Res. 3	Res. 7	Res. 15
XOR	1.0	0.13	6.78	15.42	1.0	0.03	3.35	6.54	5.44
AND / OR	more expensive than duplication - NA								
$\pm \mod n$	3.0	0.38	0.29	0.42	3.0	0.34	0.07	0.10	0.14
$\times \mod n$	21.7	NA	0.07	0.16	91.8	NA	0.00	0.01	0.02
Expansion	separate treatment (applies only to DES)								
S-Box	37.0	0.16	0.26	0.40	21.7	0.06	0.12	0.18	0.22
Word Rotation	19.3	0.38	NA	NA	4.8	0.00	0.01	NA	0.13
Word Shifting	16.6	0.35	NA	NA	4.1	0.31	0.26	0.26	0.29
Permutation	0.0	0.00	0.00	0.00	0.0	0.00	0.00	0.00	0.00
$\times \mod G(x)$	10.4	~ 0.15	NA	NA	2.6	~ 0.15	NA	NA	NA

NA = not applicable (too complex)

of check bits, a number that grows slower than the data size [28]. Moreover, EDCs could be applied to the entire data block or to parts thereof (bytes, words), allowing more flexibility in the design.

Therefore, we focus in this paper on error detection through EDCs, the basic ones being parity codes and arithmetic residue codes with the modulus 3, 7, or 15. Although all of these codes can be applied to any operation, residue codes are more suited to modular arithmetic operations, whereas parity codes are more appropriate for logical and polynomial (in $GF(2^8)$) ones. The reader interested in the formal definitions of these codes and in their prediction rules can refer to Appendix A.

The use of EDCs requires a code generator circuit (to be used initially and following every checkpoint), a set of code prediction circuits (one for each internal operation), and a comparator for checking the generated check bits against the predicted ones at each checkpoint. The scheduling of the checkpoints must also be determined and it depends on the trade-off between the desired fault coverage and the acceptable hardware and performance overheads. Since all ciphers are iterative, scheduling the checkpoints means deciding on their frequency in the round flow and on their placement either in the middle of the round or between two rounds.

All of the above listed EDCs can be applied at different levels of data granularity. Since most internal operations work on data of byte size (8 bits) or word size (16 or 32 bits) (see Table 1), these are the right levels to select from. A global code (operating on the entire data block) is not a good choice since the data block is large and is always fragmented into shorter bit sequences for processing; the prediction overhead for such a code will be too large.

To the best of our knowledge, there exists no specific and well-defined criterion for designing reliable and fault-resistant symmetric ciphers. Moreover, none of the ciphers listed in Table 1 have been implemented in a way that specifically addresses the reliability issue. We propose dealing with this issue by matching an appropriate EDC to each operation listed in Table 2 and then matching a single EDC to the whole cipher based on the operations it employs (changing EDCs within the cipher would impose a heavy hardware overhead.) Our objective is to achieve as high an error coverage as possible with as little hardware overhead as possible.

4 Complexity of EDCs for Various Operations

All of the operations in Table 2, considered in isolation, admit specific error detection codes, yet some operations are so simple and inexpensive as to allow duplication. There does not seem to exist an EDC that is inherently optimized for all of the operations. The generation and prediction techniques of the EDCs we consider for the various operations are reported in Appendix A. Based on these equations, we calculated the complexity of implementing the various EDCs (that is, generation, prediction, and comparison of the check bits) when applied to the operations listed in Table 2. The results are shown in Table 3. The hardware complexity figures in this table are normalized with respect to the complexity of an XOR gate. An EDC is considered too expensive or *not applicable* (an "NA" entry in the table) if its overhead is much larger than that of duplication. Some special cases in Table 3 are discussed next. Appendix A should be consulted for a complete justification of the comments below:

- Residue base 15 is not considered at the byte level since it uses 4 check bits—a 50 percent bit overhead—unacceptable for an EDC. It is considered for words of 16 or 32 bits, where the overhead is 25 percent and 12.5 percent, respectively.
- Both parity and residue codes are reasonable for bitwise XOR, although parity is obviously better.
- Both parity and residue codes (and any conceivable EDC) are expensive for bitwise AND and OR; they will cause an overhead larger than that of duplication.
- Both parity and residue codes are feasible for integer modular addition and subtraction. Still, residue is better suited since, for predicting the parity bits, the intermediate carries must be available, which imposes restrictions on the design of the adder/subtractor circuit. A simple ripple-carry design satisfies this requirement and, for a single byte, this design may be an acceptable solution.

- Residue code is appropriate for integer modular multiplication. Parity is expensive since all the intermediate carries must be considered and these are not generated by optimized multiplier designs.
- Expansion is used only by DES as it applies eight S-boxes of the type $6 \rightarrow 4$ to a word of 32 bits (half the data block size). Due to this fine-grained and unconventional size of the input to the S-box, any code is expensive. Further details are provided in Section 6.4.
- Both parity and residue codes incur a reasonable overhead for rotation at the byte and word level, with the exception of residue base 7, due to the fact that $7 = 2^3 - 1$ and 3 is not a divisor of either 8, 16, or 32; this is further explained in Appendix A.
- Shifting is generally expensive because the shifting amount may not be known a priori. However, we deal with only small shifting amounts (for example, shifting by 3 or 7 bits in Serpent), yielding a reasonable overhead for word parity. It is comparable to byte parity for rotation or residues for integer multiplication. The results in Table 3 apply to data-dependent shifting; hence, the actual implementation may be cheaper.
- Both parity and residue codes have reasonable overheads for permutations at the byte and word levels.
- Parity is feasible for polynomial multiplication in $GF(2^8)$ when one of the two factors is fixed (scaling); see [6]. The values reported in Table 3 apply to AES *MixColumns* and its inverse, but they have general validity.
- Residue code is expensive for polynomial multiplication in $GF(2^8)$. If the number of bits of each factor is n, then polynomial multiplication consists of XORing all of the n rows (of n bits each) of the partial product matrix and applying reduction to the most significant n bits of the result word (of $2n$ bits), which requires XORing the generator polynomial (of n bits as well) n times. In total, there are $O(n^2)$ XOR operations. Although the prediction of residue for a bitwise XOR of n bits is feasible, for n^2 bits (with $n = 8$), it is very expensive.

4.1 EDCs for S-Box

S-box is a nonlinear substitution and its treatment is, therefore, more complex than that of the other operations. S-box is usually implemented by means of a lookup table. Two kinds of faults are possible: those affecting the contents of the lookup table and those affecting the address decoder. To protect against faults in the data, the address is extended by concatenating the check bits of the input. In the entries of the lookup table for valid address code words, the corresponding correct output code words are stored (data bits plus check bits), whereas the remaining entries contain a deliberately incorrect code word (for example, 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 two sets of check bits are compared. A mismatch indicates a fault in the address decoder. When this happens, the system must output a deliberately incorrect code word, as before.

For both types of EDCs, the address is extended by concatenating the check bits. Only the parity code has an acceptable overhead since it doubles the size of the memory, resulting in 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 of 4, 8, or 16, respectively, which is unacceptable. The auxiliary memory (for detecting address decoder faults) is much smaller than the main memory and its overhead is relatively small.

Since the data bits of the deliberately incorrect output code words associated with incorrect input code words can be assumed to be constant, it is not necessary to actually store them in the lookup table as they could be generated upon request. This would reduce the size of the data section of the lookup table (but not that of the addressing section). Such a memory architecture is, however, unconventional and requires a specialized design. Moreover, with the residue code, the size of the addressing section is still larger than that with duplication. Therefore, the above conclusion regarding the applicability of only parity codes still holds.

Some ciphers use S-boxes whose content is selected arbitrarily by the original designers (for example, DES) and, hence, does not obey any apparent mathematical law. In such cases, the only way to compute the S-box is using a lookup table. Ciphers may employ a circuit for computing the S-box and a prediction circuit for the check bits could be envisioned. This circuit will be very large as it computes nonlinear functions, will heavily depend on the cipher, and is not considered here.

TABLE 4
Recommended Error Detecting Codes for Symmetric Ciphers

Cipher	Recommended Coding Techniques
Blowfish	Parity, per byte
Camellia	Intractable by EDC (use duplication)
CAST-256	Parity, per byte
DES	Parity
IDEA	Residue, but expensive
MARS	Residue, but expensive
RC5	Parity, per byte or word, or Residue
RC6	Residue
Rijndael (AES)	Parity, per byte
Serpent	Parity, per byte or four-bit nibble
Twofish	Parity, per byte

5 EDCs for Ciphers

The operations discussed above provide a full spectrum of the most common components of symmetric ciphers. We next make recommendations regarding the choice of a single EDC for each cipher, based on both Tables 2 and 3. Our recommendations are summarized in Table 4. The following comments explain some of our recommendations:

- Expansion in DES and polynomial multiplication in AES and Twofish are better suited to parity codes,

which are therefore our choice for these three ciphers.
- We choose residue codes when the cipher employs integer multiplication, such as in RC6, as integer multiplication allows only for residue prediction.
- Ciphers using S-boxes, for example, Blowfish and Camelia, require that the parity code be used since residue is more expensive than duplication.
- MARS uses both S-box and integer multiplication and, thus, has conflicting recommendations. In such a case, a more detailed overhead comparison must be performed. Having done that, we selected the residue code for MARS.
- IDEA uses only XOR, natural addition, and modular multiplication. Multiplication is modulo $(2^{16}+1)$, making the parity code a bad choice. Even residue codes are expensive since the unusual modulus results in a very complex computation of the corrective term (see (14) in Appendix A). In this case, a residue code based on the modulus 2^s (and not 2^s-1, as was considered so far) is more appropriate, leading to a simpler prediction rule for multiplication. On the other hand, such a code is less effective for the remaining operations since it protects only the least significant bits of the whole word. See Section 6.5 for further details.
- RC5 uses operations that have affordable prediction rules for both codes. Parity is better suited to XORs, whereas residues are simpler for arithmetic operations; still, both codes are good choices for RC5.

6 Checking Frequency—Single-Bit Errors

In the previous sections, we examined only the hardware overhead of the various options for detecting faults in symmetric ciphers. In this section, we take into account the error coverage of the EDCs, based on the frequency of checking for errors. All of the EDCs we described are capable of detecting 100 percent of single-bit transient faults (see, for instance, [28]) if checking for errors is performed at the end of each internal transformation of every round.

This is, however, a very high checking frequency with considerable overhead. Since the ciphers are all iterative and consist of repeating a basic round, it might be possible to perform the checking less frequently, either at the end of each round or only once, at the end of the entire sequence of rounds. This will speed up the clock rate and reduce the time latency. In a pipelined architecture, reducing the check frequency allows us to implement fewer checkers and thus reduce the hardware overhead as well.

We next show analytically that, for the single-bit error model, a single checkpoint at the end of the entire encryption procedure guarantees 100 percent coverage in two of the previously mentioned ciphers, namely, Rijndael (AES) and RC5. DES and IDEA behave differently and are analyzed separately in what follows. We selected these four ciphers since they exhibit different behaviors and are good representatives of the rest.

6.1 Analytic Model for Error Coverage

We define the *error signature* as the difference between the real values and the predicted values of the check bits for the data block. We then develop an error propagation model for each cipher and verify whether and how far the signature of an error caused by a single-bit transient fault in the middle of the encryption process propagates without disappearing throughout the subsequent steps of the encryption.

The exact error propagation model depends on the particular cipher. We derive these error propagation models based on the following assumptions:

- The key scheduling algorithm is fault free.
- The rounds following the one that has been affected by the fault are fault free and, consequently, they either *propagate* the error signature, possibly spreading it to a larger number of bits, or *cancel* it.

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.

Denote the data block to be processed by $D=[d_j]$ for $1 \le j \le m$, where d_j are the elements (bytes or words) of the data block. The error signature is represented by a vector $E=[e_j]$ for $1 \le j \le m$, satisfying $e_j = r_j - p_j$, where the bits r_j and p_j are defined as follows:

- r_j is the *real* check bit of the jth element (byte or word) of the data block.
- p_j is the *predicted* check bit of the jth element (byte or word) of the data block.

The difference operator $(-)$ depends on the algebraic structure over which the EDC is defined. In the absence of errors, $r_j = p_j$ and the null vector O is the signature corresponding to the error-free case.

The way in which the *error signature vector* is modified as it propagates throughout the subsequent rounds must be analyzed. We assume that a single transient error is injected at the beginning of the encryption round. In this case, the rounds preceding the injection of the error are, in some sense, irrelevant. However, more realistically, the error might be injected in the middle of the round. This requires the detailed analysis of one round and usually leads to the same conclusions as in the case of injecting a fault at the beginning of a round, as we have observed in [6].

In general, the prediction rule of the EDC check bits in any internal transformation of the round has the following form:

$$p'_j = f_j(p_1, p_2, \ldots, p_m, D),$$

where p_j are the predicted check bits from the preceding transformation, that is, the bits carrying the information about the presence of prior errors. Ideally, the new predicted check bits p'_j will depend only on p_j (which sometimes happens), but, in most cases, there is also a dependence on part of the data block D. Note, however, that only the dependence on the p_j is essential to allowing the propagation of the errors. If, for some special values of D (or even for all values thereof), p'_j were independent of p_j, then the propagation chain of the error signature would be interrupted and a checkpoint would have to be inserted.

By definition, the element e'_j of the error signature at the output of an internal transformation is the difference between

the values of the output check bits computed assuming that the input (predicted) check bits are correct and their values computed assuming that the input check bits are those produced by the preceding transformation. Thus,

$$e'_j = f_j(r_1, r_2, \ldots, r_m, D) - f_j(p_1, p_2, \ldots, p_m, D).$$

The above equation can be rewritten as follows:

$$\begin{aligned} e'_j = {} & f_j(p_1 + e_1, p_2 + e_2, \ldots, p_m + e_m, D) \\ & - f_j(p_1, p_2, \ldots, p_m, D). \end{aligned} \quad (1)$$

Equation (1) provides the propagation rule of the error signature. In simple cases, the right-hand side depends only on e_j (and not on p_j or D). Furthermore, it is frequently possible to separate the dependence on e_j from that on p_j and the data block D, although it is not always true and must be confirmed in every case. We stress that, should this be false, the analysis of the propagation of the error signature would be more complex.

The error signature is updated as follows: $E' = F(E)$, where F is a function determined by f_j through (1). In most cases, F is either linear or is reducible to a composition of linear functions, considerably simplifying the error signature propagation rules.

The error propagation analysis is therefore heavily dependent on the cipher. The analysis for AES is partially outlined in [6] and can serve as a guideline for the remaining ciphers.

6.2 Coverage of Single-Bit Errors in AES

We first analyze the Rijndael (AES) cipher with parity at the byte level, assuming that the key schedule procedure is error free. For a detailed description of the cipher, see [27].

The error signature of the encryption algorithm is a 4×4 matrix (with single-bit entries) and is denoted by $E = [e_{r,c}]$ for $0 \leq r, c \leq 3$.

The AES round consists of four internal transformations: *S-box* (a byte substitution), *ShiftRows* (a byte permutation), *MixColumns* (a linear transformation working independently but in the same way on each of the four columns of E), and *AddRoundKey* (addition, byte by byte, of the round key). In [6], it was proved that

- S-box maps E to itself.
- ShiftRows only permutes the entries of E.
- MixColumns maps each column of E through a linear transformation, which is the same for all four columns and which has a nonsingular coefficient matrix.
- AddRoundKey maps E to itself.

Clearly, S-box, ShiftRows, and AddRoundKey cannot cause a nonnull error signature to map to the null vector O. This is true for MixColumns as well since its coefficient matrix is nonsingular. Therefore, a nonnull error signature propagates through an AES round with no error cancellation. Thus, error checking in AES can be deferred to the end of the last round, while still achieving 100 percent coverage of single-bit transient faults.

6.3 Coverage of Single-Bit Errors in RC5

For the RC5 encryption, both the parity and the residue EDCs applied at the word level (32 bits) with a single checkpoint at the end of the last round allow detection of all single-bit errors. The proof is, however, different from that of AES since, for instance, a nonlinear behavior is encountered for the residue code. The parity and residue codes will therefore be analyzed separately. The equation describing the basic iteration of the kernel loop is

$$\begin{aligned} A_{i+1} &= ((A_i \oplus B_i) <<< B_i), \\ B_{i+1} &= ((B_i \oplus A_{i+1}) <<< A_{i+1}), \end{aligned}$$

where $1 \leq i \leq n$ is the round counter. The addition of the round keys is removed since key scheduling is assumed to be fault free, leaving XOR ($\oplus$) and data rotation ($<<<$) as the only interesting operations within our model. The detailed proof appears in Appendix B.

6.4 Coverage of Single-Bit Errors in DES

The widely used DES cipher belongs to the class of Feistel architectures in which the algorithm modifies only half of the input block at each round and then swaps the two halves. We consider here a parity EDC either at the byte or at the word level.

The operations used in DES are very difficult to deal with from the point of view of error detection. Hence, we decided to analyze the behavior of the code at three different frequencies of checking: at the end of encryption, at the end of each round, and within the round, between the S-boxes and the bit permutation. The first two are obviously interesting, whereas the third is motivated by the need to deal with bit-level permutation and byte-level parity. These three choices are discussed briefly in Section 7.

We consider two codes of different granularities, namely, byte-level parity, where four parity bits are associated with each 32-bit word, and word-level parity, with only one parity bit per word. Both schemes present some difficult issues: Although permutation preserves parity at the word level, this is not the case when implementing byte parity. The expansion (which generates 48 from 32 bits by replicating some of them) requires parity to be updated properly, taking into account the duplicated bits. Furthermore, the S-boxes are finer grained than the parity codes used; hence, it is necessary to recompute the output parity from the smaller nibbles. The following paragraphs detail the parity schemes.

6.4.1 Parity at the Byte Level

Examining Table 2, one observes that each DES round contains a permutation at the bit level. The permutation operates on 32 bits and is very irregular (see [25]). There is no simple way of predicting the four parity bits of the four output bytes generated by the permutation, starting from the four input parity bits and allowing a reasonably limited dependence on the data bits. Due to the low hardware cost of permutation, duplicating the permutation block and then comparing the outputs would require approximately the same hardware overhead as predicting the parity bits.

Therefore, the propagation chain of the predicted parity bits must be interrupted at each round iteration, soon before

executing the permutation, and a checkpoint must be inserted. The permutation is then duplicated, the outputs compared, and the parity bits regenerated soon afterward. Hence, using parity at the byte level in DES implies that checking must be done in the middle (not the end) of each round. This does not change even if parity is applied at a finer grained level, for example, at the nibble level, as shown in Section 7.

6.4.2 Parity at the Word Level

This is the simplest solution to the permutation issue; any permutation of a single word does not modify its parity. The only issues are the expansion, which needs to consider the duplicated bits, and the substitution table, where the global parity has to be recomputed from the outputs of the S-boxes. Note that using a single parity bit for an entire 32-bit word means a very low degree of fault detection, which must be balanced with more frequent checks in order to obtain an acceptable fault coverage.

6.5 Coverage of Single-Bit Errors in IDEA

IDEA is a 64-bit block cipher that uses 128-bit-long keys [23]. Its unique characteristic is that it uses only XOR, natural addition, and multiplication with the somewhat unconventional modulus $2^{16}+1$. Multiplication breaks the linearity of the cipher and acts like a 16-to-16-bit S-box [30]. This operation is computed on the fly since the size of a substitution table in memory would be prohibitive. The fact that a substitution table does not exist makes the prediction of check bits for both parity and residue codes very difficult. Instead, the check bits of the operands are verified prior to the computation of the product and, if the check is successful, the multiplication is performed and new check bits are generated. If, however, the check fails, an error is signaled or deliberately propagated (see Section 4.1).

The paradigm *check the code–compute the result–compute new check bits* was used in DES, where it was made necessary by the bit permutation at each round. However, IDEA has four multiplications, necessitating four checkpoints at every round, compared to one for DES [13], which makes prediction more expensive; moreover, since IDEA uses modular multiplication, the predicted value has to be corrected by using the most significant bits of the intermediate result. This explains the fact that the coverage of single-bit errors is not complete since the code is contaminated by the data.

6.6 Additional Considerations

The granularity of the code must be chosen carefully, depending on the operations that are actually used in the cipher. This is not only aimed at optimizing the detection rate of the code, but also at avoiding gaps in the propagation chain and, hence, delays in the computation due to excessively large-grained codes. This is especially true for ciphers using S-boxes. Since tables must be small enough to fit into the memory (unless the required values are computed on the fly, which is an exception and not the rule), they are usually targeted for 8-bit or smaller inputs. This is important when using parity at the word level or a residue code. In these cases, the approach described in Section 4.1 is not effective since we have no access to the check bits related to a single S-box input. In order to protect the S-box, we are compelled to check the correctness of the code before accessing the table, compute the correct (or the deliberately wrong) output, and, finally, generate the check bits for the output word. In summary, a larger grained code provides worse detection rates, reduces the freedom in choosing the frequency of checkpoints, and possibly increases the overall latency of the computation.

7 Checking Frequency—Multiple-Bit Errors

Since an exact analysis of the multiple-bit error case is very complicated, we present in this section the results of extensive simulation experiments we conducted, dealing with the trade-off between the frequency of checking and the error coverage for the same four ciphers, namely, AES, DES, RC5, and IDEA.

In [6], we studied the AES cipher with respect to the parity code at the byte level. We bring here only a summary of these results. We showed in Section 5 that the parity code is a natural choice due to the use of polynomial multiplication in $GF(2^8)$. It yields impressive results due to the reasonable amount of required redundancy and the high regularity and symmetry of AES. In particular, the byte-level parity code is capable of detecting all of the faults that cause an odd number of bit errors (see [8] for the proof). Those with an even number of errors are mostly detected, but complete coverage is not guaranteed, with the most common omission being two errors in the same byte, intrinsically not detectable by the parity code. An implementation of the code is described in [14].

The results of simulations with the RC5 and IDEA ciphers are described in [13] and summarized here. There were several degrees of freedom in choosing the parameters for the simulation experiments, including

- the type of code—parity or residue,
- the level of redundancy—one or four bits per word in the case of parity codes and two or four bits per word in the case of residue codes (residue modulus 3 or 15, respectively), and
- the frequency of checkpoints—at the end of each round or only at the end of the encryption.

The detection rate for multiple-bit errors is strictly dependent on the level of redundancy: the higher the redundancy, the better the detection rate. On the other hand, the percentage of undetected faults when checking at the end of each round decreases with the number of bit errors according to an exponential law (see Figs. 1 and 2). This is due to the fault model we selected, where each fault is uniformly distributed over all the rounds. Note that residue code modulo 15 is the only code that is able to detect the faults in all of the test cases since the very few fault sets that are not detectable constitute a tiny portion of all possible events. All odd-order faults are fully detected only by the round-level check of parity codes, both byte-level and word-level ones. Similar experiments were conducted for the DES cipher and the results are shown in Figs. 3 and 4. For the parity code studied, the only choices were the level of redundancy and the frequency of checkpoints. Note that using parity codes allows detecting

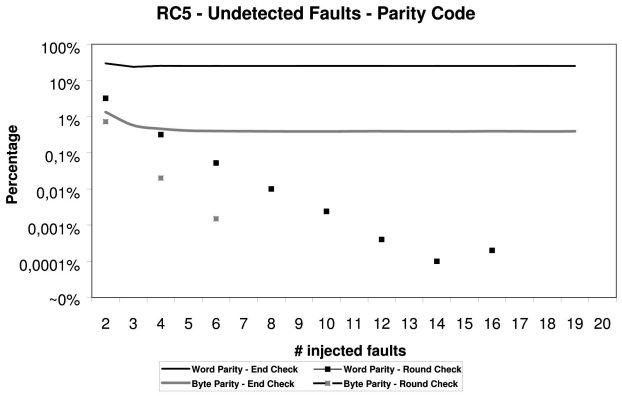

Fig. 1. Undetected faults in the RC5 encryption data path.

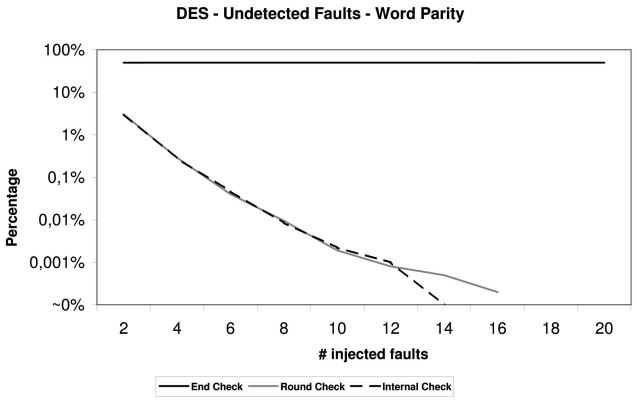

Fig. 3. Undetected faults in the DES encryption data path with word parity.

all odd-order faults. One parity bit per word gives very poor results: Checking at the end allows detecting only half of the injected faults because parity bits must be regenerated after the S-boxes anyway, even if the data is already corrupted. Moreover, there is no difference between checking at the end of each round and checking within the round. A finer grained checking gives results similar to those obtained with RC5, that is, the percentage of undetected faults decreases as the number of bit errors allowed by the model increases.

If we increase the level of redundancy by using four parity bits per word, the behavior of the parity code depends heavily on the frequency of checkpoints. Checking at the end of each round provides a good protection against even-order faults and it scales well with an increased number of injected bit errors, just like the codes in the RC5 cipher. Checking within the round allows detecting all injected faults, with the only drawback being the existence of some false positives. Finally, checking the four parity bits only at the end of the encryption results in a percentage of detected faults ranging from about 90 percent to 99 percent. Similar experiments were performed on the IDEA cipher using residue codes, which are the only choice due to the uncommon 16-bit multiplication. Checkpoints were scheduled at three different levels (as for DES), where the internal ones were located just before multiplication. The conclusions for DES hold here as well, with some interesting differences. First, only the finest grained checkpoint frequency allows us to detect all single-bit errors. Moreover, the detection percentage is poor when compared to that of AES and internal checkpoints are mandatory if we want to break the 99-percent detection barrier. Round-level checkpoints are a good compromise, whereas a single final checkpoint gives the lowest detection rate. The situation improves when using the modulus-15 residue code, but its overhead becomes as high as 25 percent, not including the additional cost due to the prediction units and comparators.

8 Conclusions

Fault attacks are becoming a serious threat to hardware implementations of ciphers and proper countermeasures must be adopted to foil them. We have presented in this paper an operation-centered approach to the incorporation of fault detection into cryptographic device implementations through the use of EDCs. Based on the operations employed in the cipher, we select an EDC (such as parity or residue code) with the least hardware overhead. We analyzed 11 ciphers and recommended an EDC for each and, for four of them (namely, AES, DES, RC5, and IDEA), we evaluated the trade-off between the checkpoint frequency and the error coverage. Although our analysis was restricted to symmetric block ciphers, our approach can be extended to public-key cryptosystems (such as RSA) as well.

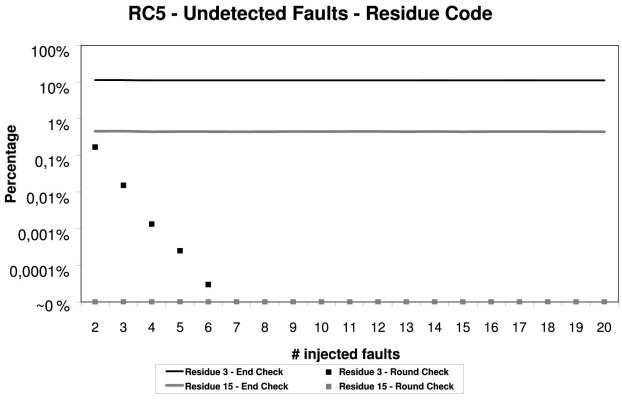

Fig. 2. Undetected faults in the RC5 encryption data path.

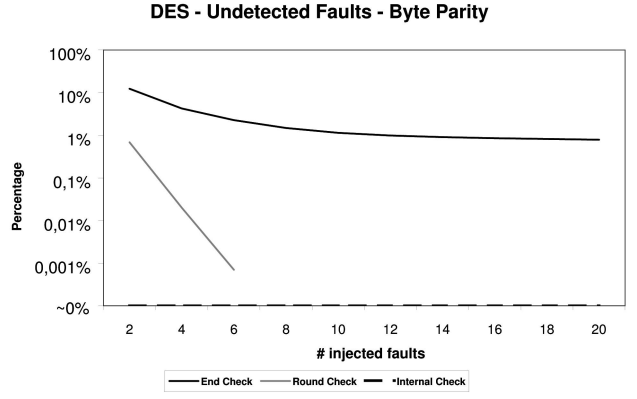

Fig. 4. Undetected faults in the DES encryption data path with byte parity.

We believe that error detecting codes can provide a useful protection against fault attacks and, in general, against errors occurring during the encryption process. They can often provide full coverage of single-bit errors and high coverage of multiple-bit errors. The actual coverage depends on many configurable parameters, such as redundancy, granularity, and validation frequency; although duplication can provide better coverage figures, it has a base overhead that is much larger than that of EDCs.

EDCs can protect against single-bit errors occurring in the data path, which are most likely benign faults and, when injected maliciously, are the most dangerous fault attacks. Moreover, EDCs can provide high coverage for multiple-bit errors, which are the most common in fault attacks. In this case, the coverage depends heavily on the fault pattern and on the redundancy. Some initial implementations have already been presented with encouraging results, showing that EDCs are a viable solution for RSA and AES. We have reported a negligible performance degradation (-3 percent) with a reasonable area overhead (33 percent) in an AES architecture. We have also embedded fault tolerance capabilities (up to 4-bit errors) with an overhead lower than duplication. Other existing countermeasures for AES have lower performance [20], lower coverage [21], or higher overhead due to the use of more complex codes.

APPENDIX A
CHECK BITS PREDICTION FOR VARIOUS OPERATIONS

This appendix provides the details of the use of parity and residue codes to detect errors. In particular, we present equations for generating the code check bits initially and equations for predicting the output check bits for each of the basic operations and data sizes in Table 2.

In what follows, we distinguish between the parity and residue codes, which are mathematically different and can be computed at different levels of granularity (byte or word), not necessarily equal to the size of the data of the internal operations. Therefore, for prediction, we distinguish between the cases when a code corresponds to m bits, but the operation operates on data of size

- m bits as well—the simple case—and
- n bits with $n > m$—a more complicated case—the granularity of operation is higher than that of the EDC.

Clearly, it does not make sense to consider a code at a coarser level of granularity than that of the operation.

A.1 Parity at the Same Level as the Operation

A single parity bit is associated with a single byte or word of $w = 8, 16, 32$ bits, depending on the cipher.

Definition. *The word parity bit $p(A)$ of A is obtained by XORing all of its w bits, that is,*

$$p(A) = p(a_{w-1} \dots a_0) = \bigoplus_{i=0}^{w-1} a_i = a_{w-1} \oplus \dots \oplus a_0, \qquad (2)$$

where a_0 (a_{w-1}) is the least (most) significant bit and w is the byte or word size.

Prediction for modular addition. It is well known (see [26] for a recent review) that, when the modulus is 2^w, 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,

$$p(A + B) = p(A) \oplus p(B) \oplus C_{in} \oplus \bigoplus_{i=0}^{w-2} C_{out}^{(i)}, \qquad (3)$$

where $C_{out}^{(i)}$ is the internal carry from the ith bit to the $(i+1)$th bit in the addition of A and B.

Prediction in modulo-2 addition. The parity of the sum is the XOR of the parities of the summands, that is,

$$p(A \oplus B) = p(A) \oplus p(B). \qquad (4)$$

Prediction in left/right rotation by $k \geq 0$ positions. Obviously, the parity of A is left unaltered,

$$p(A <<< k) = p(A), \qquad (5)$$

and similarly for right rotation.

Prediction in left/right shift by $k \geq 0$ positions. The parity of A is changed by subtracting the parity of the bits shifted out and adding the parity $p_c(k)$ of the bits shifted in, that is,

$$p(A << k) = p(A) + p(a_{w-1} \dots a_{w-k-1}) + p_c(k). \qquad (6)$$

A similar rule applies to right-shift operations.

Prediction in polynomial multiplication. When the polynomial multiplication reduces to scaling (that is, one of the coefficients is fixed), the prediction is very simple. For example, in AES, the coefficients of the *MixColumns* matrix are $01h$, $02h$, and $03h$ (in hexadecimal). The predicted values must take into account the possible polynomial reduction, which has to be performed for the most significant half of the product. The contributions from both these most significant bits and the reduction polynomial can be considered as a single correcting term. Denote $A = (a_7 a_6 \dots a_1 a_0)$, $B = (b_7 b_6 \dots b_1 b_0)$, and $\Phi = \phi_7 \phi_6 \dots \phi_1 \phi_0$, then

$$\begin{aligned} p(A \times B \mod \Phi) &= p(A) \oplus \bigoplus_{j=1}^{7} (b_j p(A >> 8 - j) p(\Phi)) \\ &= p(A) \oplus \bigoplus_{j=1}^{7} \left(b_j \bigoplus_{i=7}^{8-j} a_i p(\Phi) \right). \end{aligned} \qquad (7)$$

It can be seen that, if both operands are unknown a priori, each bit b_j must be evaluated at least once. Furthermore, the prediction rule requires that a partial evaluation of A is computed at each iteration, thus resulting in a complex design. However, if B is restricted to a small number of possibilities, then the parity prediction is simpler: AES uses the scaling coefficients $01h$, $02h$, and $03h$; Twofish uses $01h$, $5Bh$, and EFh, yielding simple prediction rules.

Prediction in permutation. For a permutation at the same level of the parity code, prediction is trivial since nothing changes:

$$A_h \mapsto A_k \Rightarrow p(A_k) = p(A_h). \tag{8}$$

A.2 Parity at a Finer Level than the Operation

Parity can be applied at the byte level in cases when the operation is applied to words of size $w = 16, 32$ bits. The propagation rules are not shown here for brevity. We only point out that most propagation rules of Appendix A.1 still hold, provided that possible external contributions are considered. These include carries between bytes or migrating bits in rotation and shifting.

A.3 Residue at the Same Level as the Operation

Definition. *The residue $r_s(A)$ of A is defined as*

$$r_s(A) = r_s(a_{31} \dots a_0) = A \mod (2^s - 1). \tag{9}$$

We focus on the residue codes for small moduli 3, 7, and 15, which correspond to $s = 2, 3$ and 4, respectively, in (9). In practice, the rules for performing arithmetic operations with the above defined residues are those of one's complement arithmetic, as taking modulo $(2^s - 1)$ is equivalent to stating that $2^s - 1 = 0$ (for example, [22]). Since this implies $2^s = 1$, (9) can be rewritten as $r_s(A) = \sum_{i=0}^{31} a_i 2^{i \bmod s} \mod (2^s - 1)$.

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

$$\begin{aligned} r_s(A + B) = &[r_s(A) + r_s(B) - c_s(A, B)] \\ &\mod (2^s - 1) \text{ for } s = 2, 3, 4. \end{aligned} \tag{10}$$

The correction term $c_s(A, B)$ takes care of the possible carry out when adding A and B as words of w bits. In fact, when $s = 2$ or 4, the arithmetic weight of the carry out is 2^w and, observing that the chosen values of s divide w (8, 16, or 32), it follows that $2^w \mod (2^s - 1) = (2^s)^{\frac{w}{s}} \mod (2^s - 1) = 1^{\frac{w}{s}} = 1$. Thus, $s = 2$ or $s = 4$ implies

$$c_s(A, B) = \text{carry out of the addition } A + B. \tag{11}$$

Prediction in modulo-2 addition. This case is more complex than natural addition. The predicted residue of the result is

$$\begin{aligned} r_s(A \oplus B) = &[r_s(A) + r_s(B) - 2r_s(A \wedge B)] \\ &\mod (2^s - 1) \quad \text{for } s = 2, 3, 4. \end{aligned} \tag{12}$$

To justify (12), observe that, when $a_i \wedge b_i = 1$, the contribution of these two bits disappears since $a_i \oplus b_i = 0$. The required correction is the negative term $-2r_s(A \wedge B) \mod (2^s - 1)$ in (12). 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 bits. Then,

$$\begin{aligned} r_s\left(A \underset{w}{<<<} k\right) &= r_s\left(\left(\sum_{i=0}^{w-1} a_i 2^i\right) \underset{w}{<<<} k\right) \\ &= r_s\left(\sum_{i=0}^{w-1} a_i 2^{(i+k) \bmod w}\right) \\ &= \sum_{i=0}^{w-1} a_i 2^{((i+k) \bmod w) \bmod s} \mod (2^s - 1) \\ &= \sum_{i=0}^{w-1} a_i 2^{(i+k) \bmod s} \mod (2^s - 1) \\ &= r_s(A) \underset{s}{<<<} (k \mod s) \end{aligned} \tag{13}$$

for $s = 2, 4$ and $w = 8, 16, 32$. The simplification

$$((i + k) \mod w) \bmod s = (i + k) \mod s$$

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

Prediction in integer modular multiplication. This case resembles the rules used in natural addition, as might be expected. The residue of the product is the product of the residues minus a correction term. Such a term is again due to the overflow of the result with respect to the size of the field; the final result must eventually be represented within the modular domain since the correction term can lead to a negative value:

$$r_s(A \cdot B) = \left[r_s(A) \cdot r_s(B) - \frac{A \cdot B}{2^w}\right] \mod (2^s - 1). \tag{14}$$

The correction term can be easily computed since the ratio is actually a w-bit shift: The w least significant bits of the product are the actual result, whereas the remaining bits are the correction term. Multipliers can be modified to provide this value at the cost of an additional output port.

Prediction in permutation. Since the permutation occurs at the same level as the residue code, the prediction rule is the same as (8):

$$A_h \mapsto A_k \Rightarrow r_s(A_k) = r_s(A_h). \tag{15}$$

A.4 Residue at a Finer Level than the Operation

The purpose of having a parity code at a finer granularity than the operation is to increase the amount of redundancy and thus improve the detection rate of the code (see Section 7). In the case of residue codes, increasing the amount of redundancy can be achieved by increasing the value of s. Using residue codes at a finer level than the operation is thus useless and can lead to unnecessary overhead and cost.

Appendix B

Proof of Complete Single Error Coverage in RC5

Let $E = [e_A, e_B]^T$ denote the initial (column) error signature associated with the register pair A, B, and, similarly, $E_i = [e_{A_i}, e_{B_i}]^T$ after $i \geq 1$ rounds.

B.1 Parity

The error propagation analysis for the RC5 encryption with parity at the word level is relatively simple. It is summarized in the following statement:

Statement 2.1. *Using parity at the word level (32 bits) for RC5 encryption and scheduling a single checkpoint at the end of the last round yields 100 percent coverage of single-bit errors.*

Proof. The predicted parity after one round has the following form, where A_i and B_i represent the values of the registers A and B (of $w = 32$ bits each) at the end of round i, with $1 \le i \le n$:

$$\begin{aligned} p(A_{i+1}) &= p\left((A_i \oplus B_i) \underset{w}{<<<} B_i\right) = p(A_i) \oplus p(B_i), \\ p(B_{i+1}) &= p\left((B_i \oplus A_{i+1}) \underset{w}{<<<} A_{i+1}\right) \\ &= p(B_i \oplus A_{i+1}) = p(B_i) \oplus p(A_{i+1}) \\ &= p(B_i) \oplus p(A_i) \oplus p(B_i) = p(A_i). \end{aligned}$$

Let p_{A_i} and p_{B_i} $(1 \le i \le n)$ be the predicted parity bits of the words A and B, respectively, at the end of round i. By applying rules (4) and (5) for predicting the parity of bitwise XOR and rotation, we obtain $p_{A_{i+1}} = p_{A_i} \oplus p_{B_i}$ and $p_{B_{i+1}} = p_{A_i}$, which can be rewritten as

$$\begin{bmatrix} p_{A_{i+1}} \\ p_{B_{i+1}} \end{bmatrix} = M \begin{bmatrix} p_{A_i} \\ p_{B_i} \end{bmatrix} \quad \text{where} \quad M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Clearly, $\det(M) = 1$, that is, the matrix M is nonsingular. Thus, using (1), we obtain

$$\begin{bmatrix} e_{A_{i+1}} \\ e_{B_{i+1}} \end{bmatrix} = M \begin{bmatrix} p_{A_i} + e_{A_i} \\ p_{B_i} + e_B \end{bmatrix} - M \begin{bmatrix} p_{A_i} \\ p_{B_i} \end{bmatrix} = M \begin{bmatrix} e_{A_i} \\ e_{B_i} \end{bmatrix}.$$

As a result, $E_n = M^n E$ and, if $E \ne O$, $E_n \ne O$ for any $n \ge 1$ since the matrix M is nonsingular. Therefore, a nonnull error signature is propagated throughout any number of rounds and can always be detected at the end of the last round. Since single-bit errors cannot cause a null error signature, their detection coverage is 100 percent. □

Note that the error propagation model for the RC5 encryption with the parity code is linear (over $GF(2)$).

B.2 Residue

The error propagation analysis for RC5 encryption with residue modulo s $(s = 2, 4)$ at word level is more complex than that for the parity code and is summarized below.

Statement 2.2. *In RC5 encryption, using the residue modulo s $(s = 2, 4)$ EDC at the word level (32 bits) and scheduling a single checkpoint at the end of the last round yields a 100 percent coverage of single-bit errors.*

Proof. The predicted residue (modulo $s = 2, 4$) after one round has the following form (use rules 12 and 13):

$$\begin{aligned} r_s(A_{i+1}) &= r_s\left((A_i \oplus B_i) \underset{w}{<<<} B_i\right) \\ &= (r_s(A_i \oplus B_i)) \underset{s}{<<<} (B_i \mod s) \\ &= [r_s(A_i) + r_s(B_i) - 2r_s(A_i \wedge B_i)] \underset{s}{<<<} (B_i \mod s), \\ r_s(B_{i+1}) &= r_s\left((B_i \oplus A_{i+1}) \underset{w}{<<<} A_{i+1}\right) \\ &= r_s(B_i \oplus A_{i+1}) \underset{s}{<<<} (A_{i+1} \mod s) \\ &= [r_s(B_i) + r_s(A_{i+1}) - 2r_s(B_i \wedge A_{i+1})] \\ &\quad \underset{s}{<<<} (A_{i+1} \mod s). \end{aligned}$$

Let r_{s,A_i} and r_{s,B_i} $(1 \le i \le n)$ be the predicted residue check bits of the words A and B, respectively, at the end of round i. Then, the above results can be rewritten as follows:

$$\begin{aligned} r_{s,A_{i+1}} &= [r_{s,A_i} + r_{s,B_i} - 2r_s(A_i \wedge B_i)] \underset{s}{<<<} (B_i \mod s) \\ r_{s,B_{i+1}} &= [r_{s,B_i} + r_{s,A_{i+1}} - 2r_s(B_i \wedge A_{i+1})] \\ &\quad \underset{s}{<<<} (A_{i+1} \mod s). \end{aligned}$$

Using (1), we obtain

$$\begin{aligned} e_{s,A_{i+1}} &= \Big[(r_{s,A_i} + e_{s,A_i} + r_{s,B_i} + e_{s,B_i} \\ &\quad - 2r_s(A_i \wedge B_i)) \underset{s}{<<<} (B_i \mod s)\Big] \\ &\quad - \Big[(r_{s,A_i} + r_{s,B_i} \\ &\quad - 2r_s(A_i \wedge B_i)) \underset{s}{<<<} (B_i \mod s)\Big], \\ e_{s,B_{i+1}} &= \Big[(r_{s,B_i} + e_{s,B_i} + r_{s,A_{i+1}} + e_{s,A_{i+1}} \\ &\quad - 2r_s(B_i \wedge A_{i+1})) \underset{s}{<<<} (A_{i+1} \mod s)\Big] \\ &\quad - \Big[(r_{s,B_i} + r_{s,A_{i+1}} \\ &\quad - 2r_s(B_i \wedge A_{i+1})) \underset{s}{<<<} (A_{i+1} \mod s)\Big]. \end{aligned}$$

The addition $(+)$ of residues is computed modulo $(2^s - 1)$; hence, rotation $(\underset{s}{<<<})$ of residues modulo s commutes with the modular addition $(+)$ of residues, that is, $(A + B) \underset{s}{<<<} C = (A \underset{s}{<<<} C) + (B \underset{s}{<<<} C)$ if A, B are two residues modulo s and C is any integer. In fact, addition modulo $(2^s - 1)$ is one's complement addition and the carry out is rotated (end-around carry). Rearranging and simplifying terms yields

$$\begin{aligned} e_{s,A_{i+1}} &= \left[e_{s,A_i} \underset{s}{<<<} (B_i \mod s)\right] \\ &\quad + \left[e_{s,B_i} \underset{s}{<<<} (B_i \mod s)\right], \\ e_{s,B_{i+1}} &= \left[e_{s,B_i} \underset{s}{<<<} (A_{i+1} \mod s)\right] \\ &\quad + \left[e_{s,A_{i+1}} \underset{s}{<<<} (A_{i+1} \mod s)\right]. \end{aligned}$$

Note that the rotation of a residue modulo s can be viewed as multiplication by a suitably chosen power of 2.

For instance, $e <<<_s A = (\alpha e) \mod (2^s - 1)$ for a coefficient $\alpha = 2^{s_A}$ with $0 \le s_A \le \lfloor \log_2(s-1) \rfloor$ (that is, the coefficient α is a power of 2 and ranges from 1 to 2^{s-1}). For $s = 2$, $\alpha = 1, 2$, whereas, for $s = 4$, $\alpha = 1, 2, 4, 8$. Hence,

$$\begin{aligned} e_{s,A_{i+1}} &= \beta_i e_{s,A_i} + \beta_i e_{s,B_i}, \\ e_{s,B_{i+1}} &= \alpha_i e_{s,B_i} + \alpha_i e_{s,A_{i+1}} = \\ &= \alpha_i e_{s,B_i} + \alpha_i(\beta_i e_{s,A_i}) + \beta_i e_{s,B_i} \\ &= \alpha_i e_{s,B_i} + \alpha_i \beta_i e_{s,A_i} + \alpha_i \beta_i e_{s,B_i} \\ &= \alpha_i \beta_i e_{s,A_i} + \alpha_i(1+\beta_i) e_{s,B_i} \end{aligned}$$

for some coefficients $\alpha_i, \beta_i = 2^{s_{A_i}}, 2^{s_{B_i}}$ with $0 \le s_{A_i}, s_{B_i} \le \lfloor \log_2(s-1) \rfloor$. This can be rewritten as

$$\begin{bmatrix} e_{s,A_{i+1}} \\ e_{s,B_{i+1}} \end{bmatrix} = M_i \begin{bmatrix} e_{s,A_i} \\ e_{s,B_i} \end{bmatrix},$$

where M_i are 2×2 matrices with entries over the integers modulo $(2^s - 1)$ (the ring of the residues modulo s):

$$M_i = \begin{bmatrix} \beta_i & \beta_i \\ \alpha_i \beta_i & \alpha_i(1+\beta_i) \end{bmatrix}.$$

Clearly,

$$\begin{aligned} \det(M_i) &= \beta_i \alpha_i (1+\beta_i) - \alpha_i \beta_i \beta_i \\ &= \alpha_i \beta_i + \alpha_i \beta_i^2 - \alpha_i \beta_i^2 = \alpha_i \beta_i. \end{aligned}$$

Thus, $\det(M_i) = 2^{s_{A_i}} 2^{s_{B_i}} = 2^{s_{A_i} + s_{B_i}}$ and it yields $\det(M_i) \neq 0$ for any $1 \le i \le n$ since exponentials cannot be zero, whatever the exponent may be.

Now, $E_{i+1} = M_i E_i$, and if, for $n \ge 1$ rounds, we denote $N_D^{(n)} = \prod_{i=1}^{n} M_i$, we obtain $E_n = (\prod_{i=1}^{n} M_i) E = N_D^{(n)} E$. Moreover, it holds that

$$\det\left(N_D^{(n)}\right) = \det\left(\prod_{i=1}^{n} M_i\right) = \prod_{i=1}^{n} \det(M_i).$$

Hence, $\det(N_D^{(n)})$ would be equal to 0 if and only if $\det(M_i) = 0$ for some i, $1 \le i \le n$, which was shown above to be impossible. It now follows that $\det(N_D^{(n)}) \neq 0$ for every $n \ge 1$ and for any initial register pair A, B, that is, for every data block D to be encrypted, and, hence, $N_D^{(n)}$ is nonsingular. Therefore, $E_n = N_D^{(n)} E \neq 0$ for a nonnull error signature E and for any $n \ge 1$.

In conclusion, a nonnull error signature is propagated throughout any number of rounds and can always be detected at the end of the last round. Since single-bit errors cannot cause a null error signature, their detection coverage is 100 percent. □

Note that, although the relationship between E_{i+1} and E_i is linear within one round (with a coefficient matrix depending on the round number), the relationship between E_n and E is not linear over $n > 1$ rounds since the matrices M_i and, consequently, $N_D^{(n)}$ depend on the round number and on the initial value of the register pair A, B and, hence, ultimately depend on the data block D to encrypt. In other words, the matrix $N_D^{(n)}$ is obtained by multiplying a finite number of matrices, the scheduling of which depends on the data block D (remember that matrix multiplication is, in general, not commutative).

This proof is a good example of how, despite error propagation being nonlinear, an analysis can still be performed.

ACKNOWLEDGMENTS

The work of Israel Koren has been supported in part by the US Defense Advanced Research Projects Agency (DARPA)/ Air Force Research Laboratory (AFRL) Network Embedded Systems Technology (NEST) program under Contract 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 of DARPA, AFRL, or the US Government.

REFERENCES

[1] C. Adams, "The CAST-256 Encryption Algorithm," Entrust Technologies, http://www.ietf.org/rfc/rfc2612.txt, 1999.
[2] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," http://www.cl.cam.ac.uk/~rja14/serpent.html, 1999.
[3] R. Anderson and M. Kuhn, "Low Cost Attacks on Tamper Resistant Devices," *Proc. Int'l Workshop Security Protocols*, 1997.
[4] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures," http://citeseer.nj.nec.com/525626.html, 2002.
[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proc. IEEE*, vol. 94, pp. 370-382, Feb. 2006.
[6] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," *IEEE Trans. Computers*, vol. 52, no. 4, pp. 492-505, Apr. 2003.
[7] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Concurrent Fault Detection in a Hardware Implementation of the *RC*5 Encryption Algorithm," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 410-419, 2003.
[8] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Detecting and Locating Faults in VLSI Implementations of the Advanced Encryption Standard," *Proc. IEEE Symp. Defect and Fault Tolerance in VLSI Systems*, pp. 105-113, 2003.
[9] E. Biham and A. Shamir, "Differential Cryptanalysis of the Full 16-Round DES," *Lecture Notes on Computer Science*, vol. 537, Springer, 1993.
[10] J. Blöemer and J.-P. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," *Proc. Seventh Int'l Conf. Financial Cryptography*, pp. 162-181, 2003.
[11] D. Boneh, R. DeMillo, and R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *J. Cryptology*, vol. 14, pp. 101-119, 2001.
[12] *Proc. Third Workshop Fault Diagnosis and Tolerance in Cryptography*, L. Breveglieri, I. Koren, D. Naccache, and J.P. Seifert, eds., 2006.
[13] L. Breveglieri, I. Koren, and P. Maistri, "Detecting Faults in Four Symmetric Key Block Ciphers," *Proc. 15th IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 258-268, Sept. 2004.
[14] L. Breveglieri, I. Koren, and P. Maistri, "Incorporating Error Detection and Online Reconfiguration into a Regular Architecture for the Advanced Encryption Standard," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems*, pp. 72-80, Oct. 2005.
[15] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS—A Candidate Cipher for AES," NIST AES Proposal, http://researchweb.watson.ibm.com/security/mars.pdf, June 1998.
[16] A.S. Butter, C.Y. Kao, and J.P. Kuruts, *DES Encryption and Decryption Unit with Error Checking*, US patent US5432848, July 1995.

[17] M. Ciet and M. Joye, "Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults," http://citeseer.nj.nec.com/ciet03elliptic.html, 2003.
[18] C. Giraud, "DFA on AES," http://citeseer.nj.nec.com/558158.html, 2003.
[19] B. Gladman, "A Specification for Rijndael, the AES Algorithm," http://fp.gladman.plus.com/, 2001.
[20] R. Karri, K. Wu, P. Mishra, and K. Yongkook, "Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 427-435, 2001.
[21] R. Karri, G. Kuznetsov, and M. Goessel, "Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers," *Proc. Fifth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '03),* pp. 113-124, 2003.
[22] I. Koren, *Computer Arithmetic Algorithms,* second ed. A.K. Peters, 2002.
[23] X. Lai and J. Massey, "A Proposal for a New Block Encryption Standard," *Advances in Cryptology—Proc. EUROCRYPT '90,* pp. 389-404, 1991.
[24] Y. Monnet, M. Renaudin, and R. Leveugle, "Designing Resistant Circuits against Malicious Faults Injection Using Asynchronous Logic," *IEEE Trans. Computers,* vol. 55, no. 9, pp. 1104-1115, Sept. 2006.
[25] Nat'l Bureau of Standards, "Data Encryption Standard," Federal Information-Processing Standard Publication No. 46, US Dept. of Commerce, Jan. 1977.
[26] M. Nicolaidis and R. Duarte, "Fault Secure Parity Prediction Booth Multipliers," *IEEE Design & Test of Computers,* vol. 16, no. 3, pp. 90-101, July-Sept. 1999.
[27] US Nat'l Inst. of Standards and Technology (NIST), "Announcing the Advanced Encryption Standard (AES)," Federal Information-Processing Standard Publication No. 197, Nov. 2001.
[28] W. Peterson and E. Weldon, *Error-Correcting Codes,* second ed. MIT Press, 1972.
[29] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad," *Proc. Fifth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '03),* pp. 77-88, 2003.
[30] H. Raddum, "Cryptanalysis of IDEA-X/2," *Proc. 10th Int'l Workshop Fast Software Encryption,* pp. 1-8, 2003.
[31] R. Rivest, "The RC5 Encryption Algorithm," *Proc. K.U. Leuven Workshop Cryptographic Algorithms,* 1995.
[32] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proc.,* pp. 191-204, 1994.
[33] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," CounterPane Labs, http://www.counterpane.com/twofish.pdf, 1998.
[34] A. Shamir, "Method and Apparatus for Protecting Public Key Schemes from Timing and Fault Attacks," US patent 5991415, 1999.
[35] C. Walter, "Data Integrity in Hardware for Modular Arithmetic," *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '00),* pp. 204-215, 2000.
[36] N. Yoshi, K. Wu, and R. Karri, "Concurrent Error Detection Schemes for Involution Ciphers," *Proc. Sixth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '04),* pp. 400-412, 2004.

Luca Breveglieri received the DrEng degree in electronic engineering and the PhD degree in electronic egineering of information and systems, in 1986 and 1992, respectively, from the Politecnico di Milano. From 1991 to 1998, he was a computer technician, network manager, and network administrator. Since 1998, he has been an associate professor at the Politecnico di Milano. His current research interests include application specific VLSI synthesis, computer arithmetic, architectures for cryptography, fault diagnosis and tolerance, and the theory of formal languages. He has approximately 60 publications in refereed journals and conferences. He was a co-guest editor for the *IEEE Transactions on Computers* special section on fault diagnosis and tolerance in cryptography in September 2006. He is cofounder of the Workshop on Fault Diagnosis and Tolerance in Cryptography (2004-2006).

Israel Koren received the BSc, MSc, and DSc degrees from the Technion-Israel Institute of Technology, Haifa, in 1967, 1970, and 1975, respectively, all in electrical engineering. He is currently a professor of electrical and computer engineering at the University of Massachusetts, Amherst. Previously, he was with the Technion-Israel Institute of Technology. He has also held visiting positions with the University of California, Berkeley, University of Southern California, Los Angeles, and University of California at Santa Barbara. He has been a consultant to several companies, including IBM, Intel, Analog Devices, AMD, Digital Equipment Corp., National Semiconductor, and Tolerant Systems. His current research interests include fault-tolerant techniques and architectures, yield and reliability enhancement, and computer arithmetic. He has published extensively in several IEEE transactions and has more than 200 publications in refereed journals and conferences. He currently serves on the editorial boards of the *IEEE Transactions on VLSI Systems*, *IEEE Computer Architecture Letters*, and the *VLSI Design Journal*. He was a co-guest editor for the *IEEE Transactions on Computers* special issue on high yield VLSI systems in April 1989, special issue on computer arithmetic in July 2000, and special section on fault diagnosis and tolerance in cryptography in September 2006, and served on the editorial board of the *IEEE Transactions on Computers* between 1992 and 1997. He has also served as general chair, program chair, and program committee member for numerous conferences. He edited and coauthored the book *Defect and Fault-Tolerance in VLSI Systems*, volume 1 (Plenum, 1989). He is the author of the textbook *Computer Arithmetic Algorithms* (A.K. Peters, 2002). He is a coauthor of the textbook *Fault Tolerant Systems* to be published by Morgan Kaufman in 2007. He is a fellow of the IEEE.

Paolo Maistri received the MS degree in electrical engineering and computer science from the University of Illinois at Chicago, the DrEng degree in computer engineering from the Politecnico di Milano in 2001, and the PhD degree in computer science from the Politecnico di Milano in 2006. He is currently a postdoctoral researcher at the Techniques of Informatics and Microelectronics for Computer Architecture (TIMA) Laboratories in Grenoble. His research interests include efficient and reliable implementations of cryptosystems, side-channel analysis, and fault attacks to public-key and symmetric ciphers.