A-SOFT-AES: Self-Adaptive Software-Implemented Fault-Tolerance for AES

Fabian Oboril, Ilias Sagar and Mehdi B. Tahoori
Chair of Dependable Nano Computing (CDNC), Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: {fabian.oboril, mehdi.tahoori}@kit.edu, ilias.sagar@student.kit.edu

***Abstract*—The Advanced Encryption Standard (AES) is one of the most widespread encryption techniques used by millions of users worldwide. Although AES was designed to withstand linear or differential attacks, the security of encrypted messages is not guaranteed. Bit flips occurring during the encryption due to runtime failures or purposely invoked by an attacker are a major security concern and can significantly jeopardize integrity, privacy, and confidentiality and hence the security of the system. Therefore, techniques to increase the reliability (fault-tolerance) and with it the security of cryptographic systems are necessary. This work proposes a self-adaptive software-implemented fault-tolerance methodology for AES (A-SOFT-AES) to enhance its fault-tolerance. This technique is based on a pool of software-implemented fault-tolerance techniques out of which it dynamically chooses the best one in terms of performance, cost, and fault-tolerance for a wide range of fault rates. Therefore, it provides superior flexibility over classic hardware-based implementations.**

I. Introduction

The amount of confidential information stored or transmitted electronically increases rapidly, making the secure storage and communication between two or more parties indispensable. To guarantee the *security* (i.e. confidentiality and integrity) of the secret information, cryptographic algorithms are used. These should ensure that nobody except the intended receiver can read or alter the information. In this context, *AES (Advanced Encryption Standard)* [10] is one of the most widespread encryption algorithms. AES is designed to withstand linear or differential cryptanalysis attacks [7] and thanks to key lengths of at least 128 bits it is also secure against brute-force attacks under current security standards. Hence, from the classical security perspective, AES is a viable choice.

However, a new class of attacks has emerged to be a severe security challenge: *Fault Attacks*. Attackers use power supply glitches, laser beams or clock perturbations to provoke bit flips in memory elements during computation and by those means try to recover used keys, decipher or alter the encrypted information [4], [5], [13], [17]. In addition also *transient runtime failures* due to natural phenomena such as radiation-induced soft errors, power supply noise or crosstalk can cause unpredictable bit flips during computations [11], [14]. Hence, these transient failures and fault attacks impair the *reliability* and security of the cryptographic system. Moreover, since fault attacks and transient effects result in bit flips, it is impossible to distinguish between them. A low fault rate can indicate radiation-induced bit flips as well as fault attacks, as some can break the system with just one bit flip. On the other hand, a high fault rate can be due to a harsh environment or a massive attack. As a result, both mechanisms have to be treated similarly. A naive approach is to suppress the output and repeat the encoding process, if faults were detected. However, in many situation this countermeasure leads to a considerable performance loss and in harsh environments it may even lead to no output at all. *Hence, the cryptographic system has to be designed fault-tolerant, to ensure a reliable operation as well as the security of the stored data or transmitted information.*

This necessitates the close interaction of security and fault-tolerance mechanisms to make the cryptographic system resilient to different types of faults. In addition performance and area costs should be as low as possible. *For this reason we present in this work a self-adaptive software-implemented fault-tolerance methodology for an AES-based system: A-SOFT-AES.* This generic approach is applicable to different AES implementations and allows to dynamically choose the best software-implemented fault-tolerance (SWIFT) technique out of a given set during runtime to maintain high performance and high fault-tolerance for a wide range of fault rates.

Compared to previous techniques, our SWIFT approach is the first one that addresses fault-tolerance in a software-implementation of AES, which is of great importance for millions of users that rely on software-based AES solutions. Furthermore, only SWIFT allows to apply an adaptive methodology, since for hardware-implementations the area overhead and hence costs for adaptive solutions are too high.

Our experimental results show that this approach achieves the best runtime results in combination with the highest fault-tolerance. It is the fastest technique for low fault rates and provides the best fault-tolerance for high fault rates, while the classical non-adaptive methods are either fast or provide a good fault-tolerance.

In summary, the contributions of this work are:

- A generic, self-adaptive SWIFT methodology for AES.
- A comprehensive performance and fault-tolerance evaluation for various fault rates and several SWIFT techniques for AES.

The rest of the work is organized as follows. In Section II the cryptographic system based on AES is introduced. Afterwards, the related work is discussed in Section III. Our A-SOFT-AES approach is explained in Section IV, where we also present a set of SWIFT techniques that are used by A-SOFT-AES. The performance and fault-tolerance evaluation can be found in Section V. Finally, we conclude the paper in Section VI.

II. The AES algorithm

In this section, the Advanced Encryption Standard (AES) is summarized. For brevity only the key aspects are addressed. Please see [7] for a more comprehensive description.

AES is a *block cipher* algorithm, which encrypts fixed-length blocks of 128 bits of plaintext into ciphertext blocks of the same length. The encrypted blocks are then linked together by using a mode of operation (e.g. concatenation).

The encryption process for a single text block is depicted in Figure 1. For this process AES uses keys that are either 128, 192 or 256 bits long, whereupon the usage of longer keys provides a better security against brute-fore attacks. Depending on the chosen key-length AES operates between $N_r = 10, 12$ or 14 rounds on a single plaintext block to transform it into a cyphertext block, as depicted in Figure 1. Therefore the encryption process interprets the text block as a 4x4-array of bytes on which linear and non-linear operations are applied. The first of these is to expand the key to gain an encryption key for each round (`ExpandKey`). Afterwards, a single

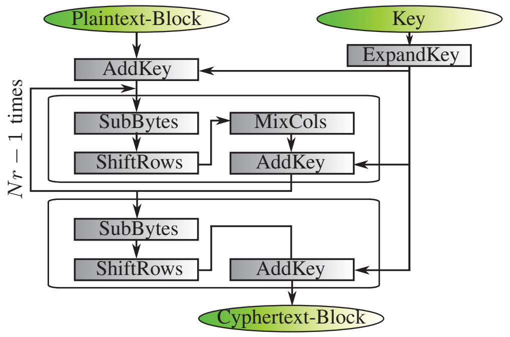


Fig. 1. Main substeps of the AES encryption flow

`AddKey` step is conducted, which adds a key to the plaintext. Then, N_r-1 rounds containing the four steps `SubBytes`, `ShiftRows`, `MixCols` and `AddKey` are executed using the expanded key. In the last round, only three of the four previously mentioned steps are performed (`MixCols` is left out) [7].

While `ShiftRows`, `MixCols` (linear diffusion layer) and `AddKey` are linear operations, `SubBytes` is non-linear. Hence, increasing the fault-tolerance of this operation is much more challenging than for the other three.

III. Related Work

For encryption algorithms, fault-tolerance against radiation-induced errors has been identified to be an important issue because of the confidentiality threat due to fault attacks [1]. In [5] it is demonstrated that the RSA key can be completely exposed by using just one erroneous signature. In [3] DES (Data Standard Algorithm) was successfully attacked by provoking random transient faults resulting in bit flips in registers. Also a non-fault-tolerant AES cannot withstand such an attack [9]. In both cases a pair of cyphertexts (a correct and a faulty one) is used to find out the last round sub-key. By this mean, other parts of the key can be derived making a brute-force attack to determine the few unknown bits easy. While these fault attacks assume that just a single bit in the encryption is flipped, the attacks presented in [4], [13], [17] make use of several bit flips. Hence, the encryption algorithms have to have fault-detection capabilities to detect and resist such attacks.

To address this challenge several hardware-based fault detection techniques have been proposed, of which we name just a few. In [12] it is proposed to decrypt the encrypted operation, round or entire block, and then to compare it with the original entity. However, this is expensive in terms of performance (if encryption and decryption are executed in serial) or hardware costs (if both are executed in parallel). Other approaches favor the usage of parity codes due to lower costs. In AES, the parity bit is easy to find for the linear `ShiftRows`, `MixCols` and `AddKey` steps [19] and also a checksum-based protection covering these three steps can be efficiently implemented [20]. In contrast, the prediction of the parity bit is not trivial for the `SubBytes` step [15]. The techniques [2], [8] propose several solutions to solve this problem. All these methods consider the SBox[1] as a 256x8 bits memory. In [2] a bit is added to the 8-bit decoder, which costs area. A cheaper solution is presented in [8] by taking advantage of the SBox invertibility, which allows to predict the input and the output parity of the SBox. Detection is done by comparing actual and predicted parities.

In this paper we use some of the previously proposed hardware-based concepts for our software-based self-adaptive approach. Therefore, we abstract the concepts of bit parity checking for `SubBytes`, `ShiftRows`, `MixCols` and `AddKey` as well as the checksum technique of [20] to software-level. In addition, our work uses a fault model, which handles several bit upsets per block encryption. This is necessary to account for fault attacks and high natural flux rates [4], [13], [17]. Instead many prior approaches such as [2], [8], [20] are based on single-bit fault models, which is no longer enough. Our self-adaptive solution covers a much wider range of fault rates.

IV. A-SOFT-AES

In this section the self-adaptive SWIFT methodology for AES is introduced. Furthermore, several SWIFT techniques are presented that are used as a pool for our adaptive solution.

A. *Self-Adaptive SWIFT-Technique*

The different characteristics of different fault detection and correction techniques in terms of performance and fault-coverage[2], drive the need for an adaptive solution to efficiently protect the system against a wide range of fault rates. Dependent on the actual fault rate such a solution should always activate the detection/correction technique that guarantees a fault-free execution with the lowest performance (i.e. runtime) overhead. The possibility to implement such a fault-tolerance technique is a great advantage of software-based approaches. In hardware, adaptive solutions would lead to a tremendous hardware overhead, which makes them infeasible.

For an adaptive technique two requirements have to be fulfilled.

1) It is necessary to have a *pool of different SWIFT techniques* that have different characteristics in terms of reliability (fault-coverage) and performance (runtime overhead), which have to be known to select the best technique for the actual fault rate. Therefore, all techniques in the pool need to be evaluated using the desired AES implementation. For our pool such an analysis is presented in Section V.
2) An *accurate fault rate estimation* during runtime of the encoding process is necessary, to allow a fault rate dependent selection of SWIFT techniques.

For the second requirement we added special counters for each operation (e.g. `SubBytes`) to the AES implementation that are incremented (by 1) every time the error recovery for the corresponding operation is invoked (see Fig. 2). Using these counters it is possible to compute the recovery rate and with this to estimate the actual fault rate. For low fault rates, when only single bit flips occur during two detection points (e.g. before and after one AES round), this approach allows to exactly calculate the actual fault rate, since in this case one recovery step corrects only a single error. Admittedly, if several bit flips occur between two detection points, the accuracy of the fault rate estimation will be lower (counters are incremented by 1, but multiple bit flips occurred). However, this can be compensated by reducing the “distance” between two detection points (adaptively), for example by checking every AES operation instead of checking an entire AES round. In fact, such an dynamic adjustment will by typically the case for increasing fault rates, since these require more detection points for a more efficient error recovery. Furthermore, it is not necessary to have an estimation accuracy of 100 %. Instead, one can take the inaccuracy of the fault rate estimation into account when defining the threshold values, which are used to select an adequate SWIFT technique for the current conditions (see Fig. 2). Using the adaptive SWIFT solution of Section V-E, the accuracy was always better than 85 % which is good enough for the mentioned purpose. A simplified version of such an adaptive SWIFT approach using three different non-adaptive techniques is shown in Fig. 2.

Please note that the described self-adaptive methodology is generic, i.e. it will work with all available software-based implementations of

[1]bijective mapping between input and output byte used by `SubBytes`

[2]fault-coverage = percentage of detected faults

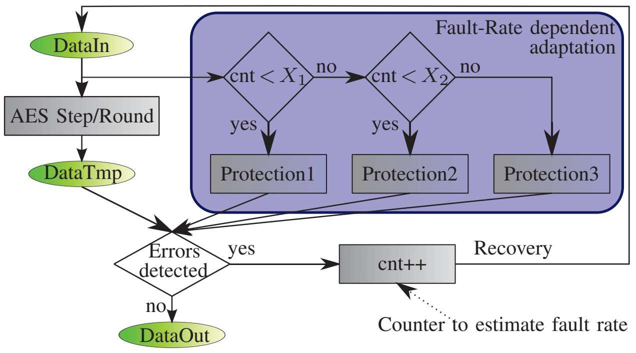


Fig. 2. Adaptive SWIFT approach for an AES step/round based on three non-adaptive error detection techniques and replay for error correction

AES. Also the pool of SWIFT techniques is not limited. Furthermore, it is very lightweight, i.e. the execution time overhead is negligible due to the low costs (a few comparisons and increments).

Limitations and Improvements: A limitation of this adaptive methodology is the fact, that a certain amount of time is required to detect and determine a change in the fault rate (reaction time). For example, a sudden, significant change of the fault rate will not be reflected immediately by the counters, which means that an inappropriate SWIFT technique can be used. This can either lead to a higher than necessary performance overhead (annoying, but acceptable) or can even cause undetected errors (unacceptable). This can be also exploited by an attacker to extract information. Therefore, the attacker fools the system first with a low fault rate, which will lead to the activation of a SWIFT technique with low fault-coverage but good performance. After a certain period, the attacker suddenly increases the fault rate. Since the system cannot adapt immediately, the fault-coverage will not be high enough to cope with all errors. This can be used to extract critical information about the system.

To reduce this risk, the adaptive technique can be improved to be more responsive, especially for increasing fault rates. Therefore, every time an error is detected, a replay is invoked which automatically uses a SWIFT technique with better fault-coverage and fault-detection capabilities. By this means the responsiveness can be massively improved and the risk of exploits or wrong results is reduced.

Furthermore, intermediate counter values can be stored and at the end of the encryption process compared to pre-defined threshold values. If a threshold is exceeded, a high temporary fault rate is indicated. To prevent wrong outputs or exploits, the entire encryption can be restarted in this case. To avoid an endless loop, a system warning can be raised to inform the user about a potential (security) problem and allowing him/her to abort the encryption.

B. Software-Based Error Detection Techniques for AES

1) Key Expansion: The key expansion is only performed once before the entire encryption process is started. Hence, the runtime of this step compared to the overall runtime is negligible (for 1 KByte of data the time for this step is just 0.1 % of the total time). Nevertheless, the `ExpandKey` step is a crucial step in terms of security and reliability. If a fault affects the result of this step, every block of the ciphertext will be wrong. Furthermore, this step also matters for fault attacks since it is often tried to manipulate parts of the expanded key, as discussed in [6]. For this reason a high fault-coverage is desired. Due to the negligible runtime we use *software-based TMR*[3] to enhance the fault-tolerance of this substep.

2) SubBytes: The `SubBytes` operation is one of the most sophisticated steps in the AES algorithm. Of course *software-based TMR* (or higher-level redundancy techniques) can be used also for the `SubBytes` operation, however here the runtime overhead is not negligible and will significantly affect the overall runtime (around 11 % overhead for the total runtime). Hence, TMR is not a viable choice for low fault rates.

[3]data triplication, execution on each of the 3 sets, bitwise majority voting

Instead a *parity-based checking scheme* can be used. However, since `SubBytes` is non-linear it is not possible to just compare the parity of the input byte(s) with the parity of the output byte(s). Instead one can exploit the way this step is implemented in software. Typically the `SubBytes` operation is implemented as lookup table (LUT), which contains the output byte for each input byte. Hence, this LUT can be extended to store also the parity relation between input and output as proposed for hardware-based approaches in [8], [19]. This information can then be used for error checking. Since the LUT is constructed just once during the initialization phase or is even pre-defined as a constant, the overhead for the additional parity information is negligible for software-based implementations.

This approach can be combined with a parity protection for `ShiftRows`, `MixCols` and `AddKey`, such that just a single parity comparison is used for detecting errors occurring in these four steps.

3) ShiftRows, MixCols, AddKey: The other three substeps of AES, `ShiftRows`, `MixCols` and `AddKey`, are linear operations. Hence, *bit-parity checking* can be used as error detection scheme.

Depending on how many bytes, rows or columns are checked together, one can adjust the fault-coverage and the performance of this approach. The slowest one, which checks the most fine-grained parities after each step is offering the best detection, while the fastest one, i.e. the one which only checks the parity of the entire 128 bits after all operations, is the worst in terms of fault-coverage. This knowledge is used by A-SOFT-AES.

An alternative approach is to use a *checksum-based error detection scheme* as presented in [20]. In this technique all round outputs are xored together to calculate a first checksum. Since each round i can be represented as $A \times B_i + K_i$ (with K_i as round key, and B_i as `SubBytes` output), the final checksum output is $\sum_i A \times B_i + K_i$. Using the distributivity law a second checksum can be derived as follows: $A \times \sum_i B_i + \sum_i K_i$. By comparing both checksums errors within `ShiftRows`, `MixCols` and `AddKey` can be detected. Furthermore, it is possible to exactly determine, which Byte is faulty.

The advantage of the checksum technique compared to the parity protection method is a much better fault-coverage at the expense of runtime. For this reason, our self-adaptive solution will use parity-based detection schemes for low fault rates and the checksum scheme for high fault rates (more details in Section V).

4) Mode of Operation: The mode of operation is a linear operation that can easily be protected by parity bits. However, for high fault rates, TMR is a better choice due to higher fault-coverage, since the runtime of this operation is negligible.

C. Software-Based Error Correction Techniques for AES

So far various detection techniques (except TMR) were discussed. The corresponding error correction methods are addressed now.

For our investigated SWIFT techniques, we always use an error correction method by means of data recovery. Hence, whenever an error is detected, some intermediate data is restored and the encryption is restarted from the intermediate checkpoint. Since we use software-based TMR for the `ExpandKey` step, there is no need for an explicit correction scheme. In case of the `Mode of Operation` we use an immediate re-execution if a bit flip was detected in this step. For the other steps we use the following solutions.

- *FullBlockRecovery*: Error checking is done at the end of each encryption pass. If an error is detected, the entire encryption pass (all N_r rounds) for the current text block is re-executed.

- *PartialBlockRecovery*: Error checking is done after each round of the encryption process. If an error is detected, the encryption process is re-started beginning with the first round of the current text block.
- *FullRoundRecovery*: Error checking is done after each round of the encryption process. If an error is detected, only the last round is re-executed.
- *PartialRoundRecovery*: Error checking is done after each round of the encryption process. If an error is detected, only the computations needed to calculate the erroneous byte are executed.

PartialRoundRecovery seems to be a very promising solution for high fault rates as it reduces the recovery overhead and hence runtime. However, for this technique much more computations are necessary compared to the other solutions to determine which byte is faulty. Hence, for low fault rates the checking overhead cannot be compensated by the low recovery overhead. In terms of checking costs, the first one is best. Again, this shows that only an adaptive technique can deliver both, high performance and high fault-tolerance.

V. Results

In this section we analyze the proposed SWIFT techniques by comparing the results of the different approaches. We perform an analysis with different fault rates between an average number of faults per encryption pass[4] of 10^{-4} and 10. Faults are injected at runtime according to the model detailed in Section V-A. The runtime for fault injection is not considered in our timing measurements.

A. Fault model and Fault Injection

Since the focus of this work is on SWIFT techniques to improve the fault-tolerance of the different AES steps, we use the basic assumption that only the operands and results of these operations can be faulty, either because the data or the computations were compromised by attacks/radiation. In contrast all data, which belongs to the control flow is correct. If these are also considered as vulnerable, one can extend the SWIFT techniques with control-flow checking approaches [16], [18] to further enhance the fault-tolerance. In summary, in this work only the results of the `KeyExpand`, `SubBytes`, `ShiftRows`, `MixCols`, `AddKey` and mode of operation can be erroneous. Faults in these steps are modeled as bit flips.

We use a fault model that allows *multiple* bit upsets per encryption pass, which is much more realistic than a single bit flip assumption [4], [13], [17]. In fact, the fault rate is variable. To achieve this we use a probabilistic model based on the assumption that all vulnerable bits have the same probability to be faulty. Therefore, the number of occurring faults is of stochastic nature. Please note that if an operation such as `MixCols` takes more time than another one, the probability of bit flips during this operation is higher.

[4]encryption pass = encryption of one text block containing 128 bit

B. Setup

We use a plaintext of 10 MByte. The key length was set to 256 bit. All experiments have been performed on a system with four 12-core AMD Opteron-6174 processors with 2.2 GHz clock rate and with a total system memory of 256 GByte of DDR3-RAM. The operating system is RHEL 6.3 and for compilation of the different solutions the build-in gcc-4.4.6 has been used with -O3 for runtime optimization. Since the fault injection methodology is of probabilistic nature, all experiments have been performed 100 times to gain reasonable data.

C. SWIFT-Analysis for `SubBytes`

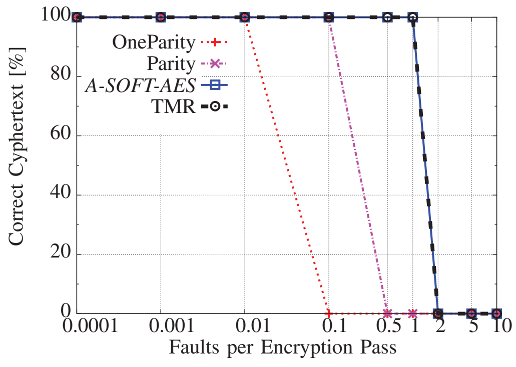
For the `SubBytes` operation the following SWIFT schemes have been investigated:

- *TMR*: Software-Based TMR.
- *Parity*: For each output byte a parity check is performed according to the description in Section IV-B2. If an error is detected the current round is re-executed (FullRoundRecovery).
- *OneParity*: The same parity checking and correction as for "Parity" is performed, except that there is just a single parity bit for all 16 Bytes.

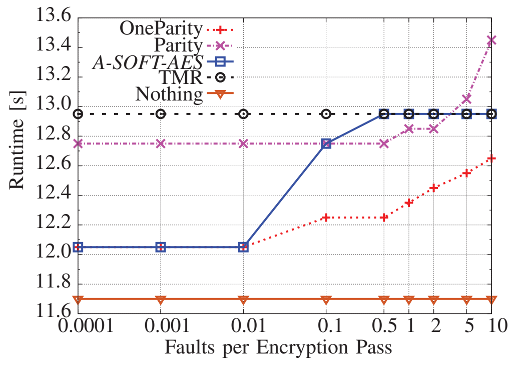
To analyze the runtime and fault-tolerance behavior of these techniques without interferences from other operations, fault injection was disabled for all steps except `SubBytes`. The results of this analysis are depicted in Figure 3. As one expects software-based TMR delivers the best results in terms of fault-tolerance at the cost of the highest runtime. However, at first glance it is surprising that even TMR cannot achieve correct results, if the average error count per text block is larger than 1. This can happen if in two of the three (intermediate) results the same bit is erroneous. Although the chance for this to happen in one single operation is very low (here: $5 \cdot 10^{-6}$), it can happen during the entire encryption run due to the total number of `SubBytes` operations (here: >9 million).

For lower fault rates TMR is not a reasonable choice, due to its runtime overhead of more than 11 %. Therefore, it is better to use parity-based solutions for lower fault rates. The fastest solution with a single parity bit (overhead is less than 3 %) can correct all faults until the average number of faults exceeds 0.01 per encryption pass. The slightly enhanced version, which protects every byte with a parity bit, can even correct 10 times more faults per second.

In the adaptive solution, we use OneParity for low fault rates (up to 0.01 faults/pass), for intermediate ones Parity is applied (from 0.01 to 0.1 faults/pass) and TMR and higher-level redundancy techniques are activated for high fault rates (> 0.1 faults/pass).

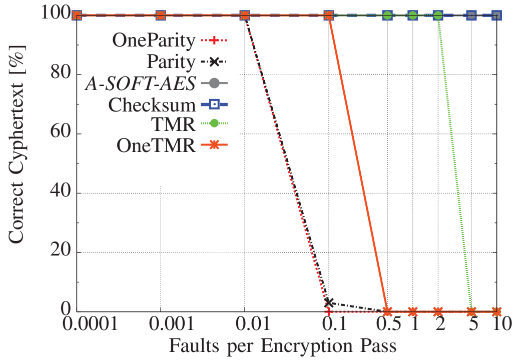


(a) Fault-Tolerance Behavior

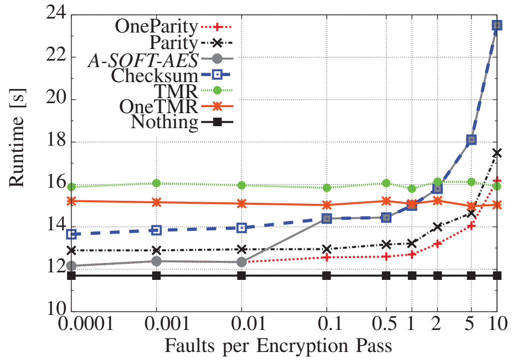


(b) Runtime Behavior

Fig. 3. Comparison of software-based fault-detection methods for `SubBytes` (OneParity: Parity for 16 Byte; Parity: Parity for each Byte; A-SOFT-AES: proposed self-adaptive solution; all use "Full Round Recovery" as correction technique)



(a) Fault-Tolerance Behavior



(b) Runtime Behavior

Fig. 4. Comparison of software-based fault-detection methods for `ShiftRows`, `MixCols`, `AddKey` (OneParity: Parity for 16 Byte with checking after all three steps; Parity: Parity for each Byte with immediate checking; OneTMR: Voting after all three steps; TMR: Immediate voting; Checksum: [20]; A-SOFT-AES: proposed self-adaptive solution; Checksum, OneParity, Parity and SWIFT-AES use "FullRoundRecovery" as correction technique)

D. *SWIFT-Analysis for `ShiftRows`, `MixCols` and `AddKey`*

For the `ShiftRows`, `MixCols` and `AddKey` operations the following SWIFT schemes have been investigated:

- *TMR*: Software-Based TMR with voting after each operation.
- *OneTMR*: Software-Based TMR with voting after all three steps.
- *Parity*: For each output byte a parity check is performed. Error correction is performed by means of FullRoundRecovery.
- *OneParity*: The same parity checking and correction as for "Parity" is performed, except that there is just a single parity bit for all 16 Bytes.
- *Checksum*: The three steps are protected by the checksum of [20]. Error correction by means of FullRoundRecovery.

Similar to the analysis of `SubBytes`, fault injection was also disabled here for steps other than `ShiftRows`, `MixCols` and `AddKey`. The results of this analysis are depicted in Figure 4. Similar to the results for `SubBytes`, software-based TMR leads to a huge performance penalty of 28 % for OneTMR and even 37 % for TMR. Moreover, as observed before neither OneTMR nor TMR can achieve correct encryption results under high fault rates, due to the same reason as for `SubBytes`. In contrast the Checksum scheme can detect all faults even for the highest investigated fault rates. However, also the Checksum scheme has a relatively high performance penalty of 17 % for low fault rates. Hence, our self-adaptive A-SOFT-AES implementation uses a parity-based fault-tolerance approach (OneParity with an overhead of 3 %) for low fault rates (up to 0.01 faults/pass), and in all other cases applies the Checksum method.

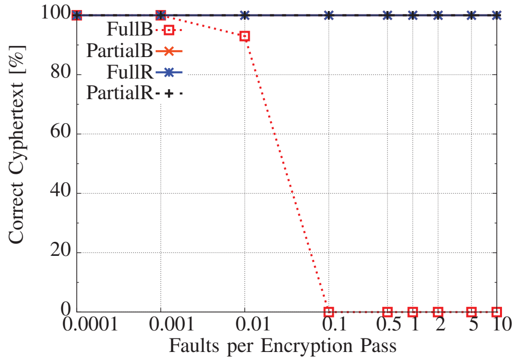
E. *Error Correction Analysis*

For all results presented so far, FullRoundRecovery was applied as correction scheme. The reason for this choice is depicted in Figure 5, for which we used the Checksum fault-detection scheme and injected faults into the `ShiftRows`, `MixCols` and `AddKey` operations. Obviously, the techniques (see Section IV-C for details) in which more than one round is re-executed (i.e. FullBlockRecovery and PartialBlockRecovery) have a huge runtime overhead for high fault rates. This is due to the fact that also fault-free rounds may have to be repeated. Moreover, FullBlockRecovery cannot maintain correct results for high fault rates, since it can happen that the same bit is flipped in two different rounds and hence remains undetected during error checking. Although PartialRoundRecovery sounds great in theory, as only very few operations have to be re-executed if a fault is detected, it performs worse than FullRoundRecovery. The reason for this behavior is that for PartialRoundRecovery more data has to be calculated and checked to find the faulty bit, which increases the runtime overhead (34 % instead of 17 %). In addition, if several bits are faulty the 1-by-1 recalculation takes more time than to recalculate all 128 bits at once. In summary, the best correction solution in terms of runtime and fault-tolerance is FullRoundRecovery.

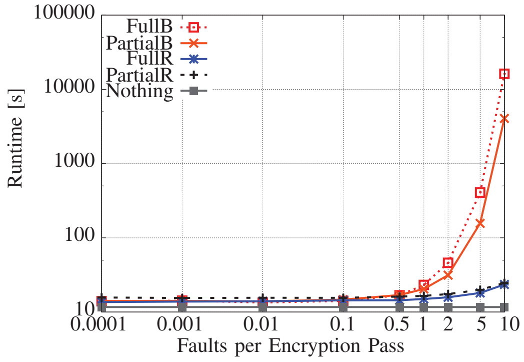
F. *Putting Everything Together*

As we have analyzed the different detection and correction schemes, we put everything together and investigate the entire encryption process including `ExpandKey` and the `Mode of Operation` (here: `Counter mode (CTR)`[5]). As explained in Section IV-B1

[5]A random vector (RV) is chosen. For each text block RV is incremented by 1 and then encrypted. The plaintext is xored to the encrypted result.

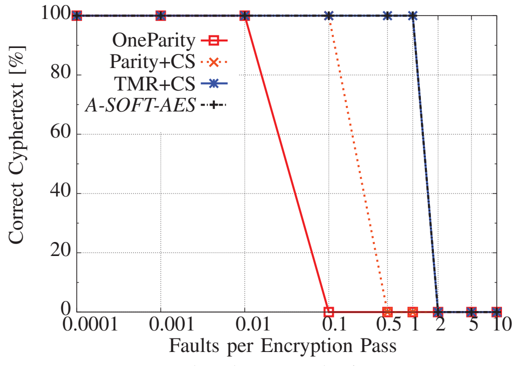


(a) Fault-Tolerance Behavior

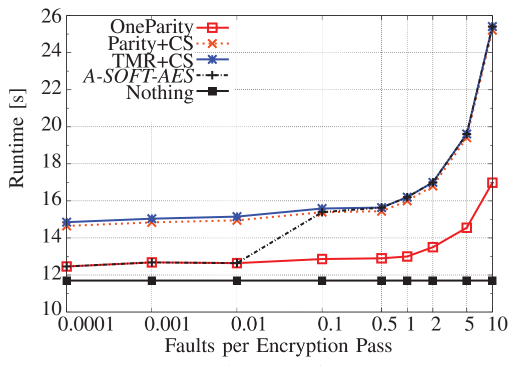


(b) Runtime Behavior

Fig. 5. Comparison of software-based error correction schemes (FullB: FullBlockRecovery; PartialB: PartialBlockRecovery; FullR: FullRoundRecovery; PartialR: PartialRoundRecovery as described in Section IV-C)



(a) Fault-Tolerance Behavior



(b) Runtime Behavior

Fig. 6. Comparison of different SWIFT techniques for AES (Parity+CS: Parity for `SubBytes`, Checksum for `ShiftRows, MixCols, AddKey`, TMR for `ExpandKey, CTR`; TMR+CS: Parity for `ExpandKey, CTR, SubBytes`, Checksum for `ShiftRows, MixCols, AddKey`; A-SOFT-AES: proposed self-adaptive technique; OneParity: OneParity for `SubBytes, ShiftRows, MixCols, AddKey`, Parity for `CTR` and TMR for `ExpandKey`)

Operation	Protection	Fault Rate	Runtime Overhead
`ExpandKey`	TMR	up to 10 faults/block	0.0 %
`CTR`	Parity	up to 1 faults/block	0.1 %
	TMR	up to 10 faults/block	0.3 %

TABLE I
SWIFT-TECHNIQUES FOR `EXPANDKEY` AND `CTR` MODE THAT PROVIDE CORRECT ENCRYPTION RESULTS IN 100 % OF THE RUNS

`ExpandKey` will be protected using software-based TMR, and the `CTR` mode is protected with a parity bit for each byte for low fault rates and by TMR for high fault rates (see Table I). Using these techniques all errors in `ExpandKey` as well as in the `CTR` mode were detected and corrected with negligible performance costs.

The runtime and fault-tolerance behavior of our self-adaptive A-SOFT-AES is depicted in Figure 6. As one can see A-SOFT-AES is always the fastest and most fault-tolerant technique over a wide range of fault-rates (from 0.0001 upto 1 fault/pass). Moreover, the runtime overhead due to the adaptive engine is negligible compared to non-adaptive solutions. However, even A-SOFT-AES cannot guarantee a fault-free execution. For example, for fault rates higher than 1 fault/pass the software-based TMR method for `SubBytes` cannot mask all faults leading to erroneous encryption results. If fault-tolerance for such fault rates is desired, one needs to add even higher levels of modular redundancy such as 5MR to the SWIFT technique pool that is used by A-SOFT-AES.

VI. CONCLUSION

The AES algorithm is one of the most widespread and secure encryption techniques available today. However, fault attacks and bit flips due to environmental disturbances are major challenges and can significantly threaten the system security and reliability. Therefore, techniques to increase the reliability and hence the security of cryptographic systems are necessary.

This work presents the first self-adaptive, software-implemented fault-tolerance approach for AES: A-SOFT-AES. It is based on a variety of different detection and correction techniques such as parity-protection, modular redundancy methods or sophisticated checksum schemes out of which it chooses the best performing one that still allows a fault-free encryption. Our experimental results show that A-SOFT-AES minimizes the performance costs while guaranteeing high fault-tolerance for a wide range of fault rates, which is impossible to achieve with classical, non-adaptive solutions.

REFERENCES

[1] A. Barenghi *et al.*, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov. 2012.
[2] G. Bertoni *et al.*, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," *IEEE Trans. on Computers*, vol. 52, no. 4, pp. 492–505, Apr. 2003.
[3] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Proc. of the 17th Annual Int'l Cryptology Conf. on Advances in Cryptology*, pp. 513–525, 1997.
[4] J. Bloemer and J. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," in *Computer Aided Verification*, vol. 2742, pp. 162–181, Aug. 2003.
[5] D. Boneh *et al.*, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, pp. 101–119, Jan. 2001.
[6] C.-N. Chen and S.-M. Yen, "Differential Fault Analysis on AES Key Schedule and some Countermeasures," in *Proc. of the 8th Australasian Conf. on Information Security and Privacy*, pp. 118–129, 2003.
[7] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, 1st ed. Springer Berlin / Heidelberg, Feb. 2002.
[8] G. Di Natale *et al.*, "A Novel Parity Bit Scheme for SBox in AES Circuits," in *Design and Diagnostics of Electronic Circuits and Systems*, pp. 1–5, April 2007.
[9] P. Dusart *et al.*, "Differential Fault Analysis on A.E.S," in *Applied Cryptography and Network Security*, vol. 2846, pp. 293–306, Oct. 2003.
[10] N. FIPS, "Announcing the Advanced Encryption Standard (AES)," *Information Technology Laboratory, National Institute of Standards and Technology*, vol. 5, no. 4, 2001.
[11] B. Gill *et al.*, "Comparison of Alpha-Particle and Neutron-Induced Combinational and Sequential Logic Error Rates at the 32nm Technology Node," in *IEEE Int'l Reliability Physics Symp.*, pp. 199–205, Apr. 2009.
[12] R. Karri *et al.*, "Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture," pp. 427–435, Oct. 2001.
[13] C. Kim and J.-J. Quisquater, "New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough," in *Smart Card Research and Advanced Applications*, vol. 5189, pp. 48–60, 2008.
[14] N. N. Mahatme *et al.*, "Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process," *IEEE Trans. on Nuclear Science*, vol. 58, no. 6, pp. 2719–2725, Dec. 2011.
[15] V. Ocheretnij *et al.*, "On-Line Error Detection and BIST for the AES Encryption Algorithm with Different S-Box Implementations," in *Proc. of the 11th IEEE Int'l On-Line Testing Symp.*, pp. 141–146, 2005.
[16] N. Oh *et al.*, "Control-flow checking by software signatures," *Reliability, IEEE Trans. on*, vol. 51, no. 1, pp. 111–122, 2002.
[17] D. Saha *et al.*, "A Diagonal Fault Attack on the Advanced Encryption Standard," *IACR eprint archive*, vol. 581, pp. 293–306, Oct. 2009.
[18] R. Vemu *et al.*, "Acce: Automatic correction of control-flow errors," in *Test Conf., 2007. ITC 2007. IEEE Int'l*, pp. 1–10, 2007.
[19] K. Wu *et al.*, "Low Cost Concurrent Error Detection for the Advanced Encryption Standard," in *Proc. of the Int'l Test Conf. on Int'l Test Conf.*, pp. 1242–1248, 2004.
[20] C. Zhang *et al.*, "An Algorithm Based Concurrent Error Detection Scheme for AES," in *Cryptology and Network Security*, vol. 6467, pp. 31–42, Nov. 2010.