# Low-Cost Software Countermeasures Against Fault Attacks: Implementation and Performances Trade Offs

Alessandro Barenghi
DEI – Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
barenghi@elet.polimi.it

Luca Breveglieri
DEI – Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
brevegli@elet.polimi.it

Israel Koren
Department of Electrical & Computer Engineering
University of Massachusetts
Amherst MA 01003, USA
koren@ecs.umass.edu

Gerardo Pelosi
DEI – Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5, 20133
Milano, Italy
pelosi@elet.polimi.it

Francesco Regazzoni
UCL Crypto Group, Université Catholique de Louvain
ALaRI - University of Lugano
Via Buffi 13, 6900 Lugano, Switzerland
regazzoni@alari.ch

## ABSTRACT

In this paper we present software countermeasures specifically designed to counteract fault attacks during the execution of a software implementation of a cryptographic algorithm and analyze their efficiency. We propose two approaches that are based on the insertion of redundant computations and checks, which in their general form, are suitable for any cryptographic algorithm. In particular, we focus on selective instruction duplication, employed to detect single errors, instruction triplication to support also error correction, and parity checking to detect corruption of a stored value. We developed a framework to automatically add the desired countermeasure, and we support the possibility to apply the selected redundancy to either all the instructions of the cryptographic routine or restrict it to the most sensitive ones, such as table lookups and key fetching. Considering an ARM processor as a target platform and AES as a target algorithm, we evaluate the overhead of each proposed countermeasure while keeping the robustness of the implementation high enough to thwart most or all the known fault attacks. Experimental results show that in the considered architecture, the fastest solution is per-instruction selective doubling and checking, and that the instruction triplication is a viable alternative if very high levels of fault resistance are required.



## Categories and Subject Descriptors

C.3 [**Special-Purpose and Application Based Systems**]: Microprocessor/microcomputer applications; C.5.3 [**Computer System Implementation**]: Microcomputers—*portable devices*; E.3 [**Data Encryption**]: Standards (AES)

## General Terms

Security

## Keywords

Side-Channel Attacks, Fault Attacks

## 1. INTRODUCTION

Embedded systems now constitute the largest segment of the electronic consumer market. Such a position was gained thanks to their diffusion in our everyday life, in diverse applications such as fuel injection in cars, access control systems and smart-cards. Many of these applications require the use of cryptographic algorithms to secure the data that they process. The widespread use of security-sensitive embedded systems brings up new design challenges. Although the traditional design objectives such as power consumption, memory usage, real time performances and reconfigurability continue to be important, the use of embedded systems for critical functions makes security one of the most significant requirements of the system design.

Among the possible attacks proposed in the past, the ones which target the physical implementation of the cryptographic algorithm are the most dangerous, since they are often easy enough to be successfully carried out. Fault injection attacks, in particular, have proved to be a very effective and relatively inexpensive way to retrieve the secret information processed by electronic devices. In a typical fault attack scenario, the adversary injects a number of faults during the computation of a cryptographic routine and then analyzes the faulty outputs to derive the secret key of the cipher.

Recent works have improved this technique by significantly reducing the effort required to mount a fault injection attack, thus raising the threat level posed by such attacks.

Section ?? provides a description of the threat model and gives a brief summary of the AES block cipher chosen as a case study. Section ?? describes the proposed countermeasures, and Section 5 reports the results of the experimental campaign conducted in order to ascertain the effectiveness of the described methods. Finally, Section 6 presents our conclusions.

# 2. PRELIMINARIES

In this section we briefly describe the possible threats posed to cryptographic algorithms running on an embedded device and present the cipher of choice for the validation of our proposed countermeasures.

## 2.1 Threat Model

Recent research [3, 10] has provided key insights into how to induce faults through the use of reasonably cheap equipment and simple workbenches to properly vary the feeding voltage of an ARM based device, thus resulting in very regular corruptions of the values loaded from the memory. This kind of fault can be injected with almost no knowledge about the implementation details of the device and thus should be regarded as particularly dangerous. Through the use of more sophisticated workbenches (e.g., laser injection devices [11] or timed EM-pulses [1]) it is also possible to inject errors into a device in a precise way. This kind of attack, albeit as realistic as the previous one, requires a higher level of technical knowledge and more expensive equipment to be practically put into action and thus is regarded as realistically applicable only in a scenario where highly valuable goods are at stake. In [2] it was shown that widely deployed embedded processors can be attacked by judiciously lowering the processor's supply voltage. The induced faults can be injected with very high precision to affect instructions which load data from memory and, through exploiting the resulting faulty ciphertexts, the secret key of an AES cipher can be easily inferred.

## 2.2 AES Overview

In this paper we explore countermeasures against such fault attacks with a tunable level of protection in order to provide a full spectrum of the trade-off between the level of protection and the implementation overhead. We focus here on the AES cipher [5], which is employed in a wide range of devices.

The AES cipher executes a number of round transformations on the input plaintext where the output of each round is the input to the next one. In contrast to Feistel networks, AES encrypts the whole input at each round, and consequently, it employs a comparably small number of rounds. The number of rounds $r$ is determined by the key length: 128-bit uses 10 rounds, 192-bit 12 and 256-bit 14. In software, AES can be implemented using only bitwise `xor` operations, table-lookups and 1-byte shifts [5]. Each round is composed of the same steps, except for the first where an extra addition of a round key is inserted, and the last where the (MixColumns) operation is skipped. Each step operates on 16 bytes of data (referred as the internal *state* of the cipher) generally viewed as a 4×4 matrix of bytes or an array of four 32-bit words, where each word corresponds to a column of the *state* table. The four round stages are: AddRoundKey (`xor` addition of a scheduled round key), SubBytes (byte substitution by a lookup table ($S$-box)), ShiftRows (cyclical shifting of bytes), and MixColumns (linear transformation which mixes column state data). Given the cipher key $k$, the KeySchedule procedure outputs $r+1$ round subkeys, with each subkey being 16 byte wide. Algorithm 2.1 shows the complete encryption process.

**Algorithm 2.1**: AES Encryption

**Input**: $p$, plaintext block; $k$, cipher key
**Output**: $c$, ciphertext block

**begin**
  $\langle k^{(0)}, k^{(1)}, \ldots, k^{(r)} \rangle \leftarrow$ KeySchedule($k$)
  $c \leftarrow$ AddRoundKey($p,\ k^{(0)}$);
  **foreach** $i \in \{1, \ldots, r\}$ **do**
    $c \leftarrow$ SubBytes($c$)
    $c \leftarrow$ ShiftRows($c$)
    $c \leftarrow$ MixColumns($c$)
    $c \leftarrow$ AddRoundKey($c,\ k^{(i)}$)
  $c \leftarrow$ SubBytes($c$)
  $c \leftarrow$ ShiftRows($c$)
  $c \leftarrow$ AddRoundKey($c,\ k^{(r)}$)
  **return** $c$
**end**

The enciphering procedure is amenable to several software implementations which trade-off memory and computational resources in order to obtain the best performance for the specific architecture.

Specifically, the different steps of the round transformation can be combined into a single set of table lookups, allowing for very fast implementations on processors having word length of 32 bits or more [5]. Denote by $a_{i,j}$ the generic element of the state table, by $a$ the generic value of a byte variable, by $S[0, \ldots, 255]$ the 256-bytes of the $S$-box table and by $\circ$ a $GF(2^8)$ finite field multiplication [5]. Let $T_0$, $T_1$, $T_2$ and $T_3$ be four lookup tables, each viewed as a 256 sequence of 32-bit words, containing results from the combination of the round operations as follows:

$$T_0[a] = \begin{bmatrix} S[a] \circ 02 \\ S[a] \\ S[a] \\ S[a] \circ 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a] \circ 03 \\ S[a] \circ 02 \\ S[a] \\ S[a] \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ S[a] \circ 03 \\ S[a] \circ 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \circ 03 \\ S[a] \circ 02 \end{bmatrix}$$

These tables are used to compute the round stages operations as a whole, as described by the following equation, where $k_j$ is the $j$-th word of the expanded key and $A_j = \langle a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j} \rangle$ is the $j$-th column of the state table considered as a single 32-bit word (with the simplified notation: $A_j = A_{j \bmod 4}$, $a_{i,j} = a_{i,j \bmod 4}$):

$$A_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus k_j$$

The four tables $T_0$, $T_1$, $T_2$ and $T_3$ (called *T-tables* from now on) use 4KB of storage space and their main goal is to avoid performing the MixColumns and InvMixColumns

transformations as these operations, in the original definition of Rijdael algorithm, perform Galois Field multiplication by fixed constants which map poorly to general purpose processors in terms of performance.

Notably, in the final round of the encryption there is no MixColumns operation, and also the KeySchedule algorithm requires pure substitution operations. Whilst these facts could represent an impairment in the use of $T$ tables, it is possible to extract efficiently the $S$ table through proper masking of the $T$ tables.

Since the $T$-tables may be derived also through rotating each word of $T_0$ by $i$ bytes, $T_i[a] = \text{RotByte}(T_0[a], i)$, $i \in \{0, \ldots, 3\}$ in order to reduce the active memory footprint used within each round, each column of the state table may be also computed as:

$$A_j = T_0[a_{0,j}] \oplus \text{RotByte}(T_0[a_{1,j-1}], 1) \oplus$$

$$\oplus \text{RotByte}(T_0[a_{2,j-2}], 2) \oplus \text{RotByte}(T_0[a_{3,j-3}], 3) \oplus k_j$$

This variation reduces the lookup tables to a single 1KB one, thus lowering the burden on the caches, while incurring a penalty of only three extra rotations per column per round with respect to the 4 $T$-table implementation.

Decryption requires different tables from those used by the encryption, therefore, an AES implementation able to perform both encryption and decryption may require up to 8KB of memory, which may extend to 16KB if the last round operations are realized with ad-hoc tables.

When employing general-purpose processors, endowed with wide D-caches, the $T$-table implementation is more effective since the memory access latency is lower than the computation time that would be required in place of each $T$-table lookup. On the other hand, in cache constrained environments a valid alternative to the use of $T$-tables is the computation of the entire AES rounds on the processor, memorizing only the $S$-box and the inverse $S$-box tables needed to perform the substitution operations.

Another downside of employing data caches,regardless of the constrains on the size of the chip is represented by cache timing attacks to cryptographic algorithms . This kind of attack is able to infer from the loading times of a value its position in the main memory, and thus obtain an information on the memory access patterns of a cipher. This in turn has been demonstrated to be enough to break AES in [13], and thus the employment of cache should be either avoided or carefully regulated when during the computation of this algorithm.

# 3. TARGET ARCHITECTURE

The target architecture to validate our countermeasures is the ARM family of processors. Our proposed countermeasures are employable on every ARM processor starting from ARM7 (ARMv3 architecture). This architecture is a load-store RISC machine, where all the instructions are executed in one clock cycle, and may be conditioned by a flag of the program status word. This feature was introduced in order to compensate for the missing branch predictor which was introduced only later on in ARM8.

The `load` instructions are not bound to be executed in a single cycle, instead, they may stall the pipeline until the information is retrieved from the main memory. The memory latencies have a broad range due to the wide field of applications in which the ARM processor is deployed.

The RISC architecture and the fully predicated instructions (which are used in place of small decisional constructs) made the ARM architecture suitable for the use in low power or memory constrained environments. This has made them dominant in the mobile and embedded electronics market as relatively low cost and small microprocessors and microcontrollers. Thanks to its high power efficiency, the ARM architecture has been lately employed also in mobile multimedia enabled platforms. This, in turn, has lead to a rising computing power and memory demand towards the platform, leading to the formation of the same CPU-Memory gap experienced in high end microprocessors.

In order to cope with the raising demands of computing performances, the latest ARM based platforms are split, according to the target use, into two families, Cortex-A and Cortex-M, while retaining full binary compatibility. Cortex-A based platforms are targeted to high end mobile multimedia devices and are often endowed with two levels of caches to cope with the load latency from high capacity, but slow, off-chip memory. The L1 cache is split into two halves, which are used as dedicated instruction and data cache, fitting the Harvard architecture of the ARMv7 platform, while the L2 cache is unified for both data and instructions. The Cortex-M family is targeted to the microcontroller and highly energy constrained environments. For this cpu family, the Cortex-M reference manual from ARM [**?**] suggests the employment in the design of tightly coupled memories: i.e. small amounts of SRAM integrated on the same die of the microprocessor and with very fast access latencies (one or two clock cycles). Embedding the memories in full on the same chip saves the need to employ caches since the whole main memory is actually implemented with cache-like timing features and thus avoids the need for the power consuming combinatorial logic driving the caches. While typical memory access latencies for chips having on-die integrated memory range from one to two clock cycles, off chip DRAM memories tend to require up to 50 CPU clock cycles to fetch the data.

The ARM architecture has 16 general-purpose 32-bit registers. Although the architectural specification does not impose any restriction on their usage, the standard ARM ABI interface mandates the last three registers (`r13-r15`) to be used to keep the context of the running program (stack pointer, link register and program counter). Moreover, the content of `r12` is not guaranteed to be preserved among function calls, thus acting as a scratch register.

In order to provide fast shifting and rotation of loaded values, one of the two loading lines for the ALU has a 32-bit barrel shifter able to act without delaying the loading of the value from a register into the ALU. This implies that it is possible to perform computations directly with a shifted or rotated operand without losing an extra clock cycle.

# 4. COUNTERMEASURES

In this paper we also explore the incorporation of error detecting and correction techniques into the AES algorithm implemented in software and executed on the ARM architecture without modifications to either the ABI or the underlying architecture.

We consider three ways of inserting redundant computations and checking: the use of instruction doubling to detect errors (Dual Modular Redundant (DMR) technique), instruction triplication to add error correction capabilities

(Triple Modular Redundant (TMR) technique), and the computation of parity bits and checking them against stored values (Parity (PAR) technique).

These countermeasures can be either applied to the whole AES algorithm or only to the parts sensitive to known attacks (Selective Insertion) in order to reduce the computational overhead. The advantage of the former technique is the greater ease of introduction, since the insertion of the countermeasure can be done directly into the code obtained from disassembling the executable object code, without knowing any details about the AES implementation, nor the need to take into account the optimization strategies of the compiler to prevent the removal of the redundant operations. The selective insertion technique requires some knowledge of the enciphering program structure in order to locate the sensitive parts which need the application of the countermeasure.

## 4.1 Instruction Duplication

The first method (DMR at instruction level) consists of duplicating the execution of any (or a specific) instruction and storing the result into a different register of the CPU. This is possible in our implementation since the AES algorithm uses only 9 registers (in our binary `r1`–`r9`) in the ARM architecture, thus leaving 4 free for our purposes (`r0`, `r10`–`r12`). After repeating the instruction the results are compared and, in case a mismatch is detected the inserted code jumps to an error management routine which may either signal the error or fill the state of the computation with random numbers to avoid information leakage. The following code sample illustrates the insertion of the countermeasure to protect the load of a value in `r4` from a memory location whose address is contained in `r7`:

```
1. ldr r4,[r7];
2. ldr r12,[r7];
3. cmp r12,r4;
4. bne <error>;
```

This kind of countermeasure fails to detect two kinds of faults: two identical faults injected in the each of the payloads of the load operations and a single fault injected in a load and an instruction skipping fault which allows to bypass the branch instruction after the comparison.

## 4.2 Instruction Tripling

The second method (TMR) implements a full triple modular redundancy scheme on an instruction through repeating it three times and storing the two extra results into two free registers. In the following code snippet the protected instruction is an exclusive-or (`eor`) between `r1`, `r2` with result stored in `r4`:

```
 1. eor r12,r12,r12;
 3. eor r10,r1,r2;
 4. eor r0,r1,r2;
 2. eor r4,r1,r2;
 5. cmp r4,r10;
 6. eoreq r12,r12,#1;
 7. cmp r4,r0;
 8. eoreq r12,r12,#2;
 9. cmp r10,r0;
10. eoreq r12,r12,#4;
```

```
11. cmp r12,#0
12. beq <error>;

13. cmp r12,#4;
14. moveq r4,r0;
```

This technique employs a fourth register, `r12` to record the effects of the correctness checks (lines 5–10) and determine whether to correct a single error (lines 13–14), detect two errors (lines 11–12) or leave the result untouched. The `eor` operation between `r1`, `r2` is performed thrice and the three results are stored in the target register (`r4`) and in two scratch pad registers (`r10`, `r0`), respectively. The algorithm then proceeds to check the equality of the values pairwise and stores the result as a single bit flag in register `r12` which was zeroed at the beginning. To save comparison instructions, it is possible to check only if either the first value (which has already been loaded in the target register), is faulty or not and eventually correct it, without taking care to correct the values in the two scratch pad registers.

The minimum effort required by an attacker to thwart the instruction tripling countermeasure is represented by the insertion of two perfectly identical faults into two of the three sensitive operations. This in turn implies that the attacker must be able to damage two instructions , which are only a single clock cycle apart, in exactly the same way. Another alternative to this is to damage the operation which will be actually setting the correct output register (i.e. not one of the redundant ones) and subsequently skip both the branch to error condition and the last moveq instruction which would restore the correct result anyways. This would mean being able to inject one data altering and two instruction skip faults within a 11 clock cycles timeframe, with the middle fault being 2 clock cycles afar from the last.

## 4.3 Parity Checking

The third technique considered is the employment of a tabulated parity bit in order to check the consistency of the loaded values. This technique is not applicable to generic arithmetical/logical operations and will therefore leave the computational instructions unprotected. Another disadvantage of employing parity in software is the fact that the parity bit related to each protected value must be both computed from the word that contains it, at the expense of additional computation steps, or stored in a very sparse representation (one bit per byte or one bit per word, depending on the architecture alignment). The following code snippet represents the most straightforward way to compute the parity bit of a value contained in `r4` in order to check it against a precomputed value stored in memory at the address contained in `r7`. The protected value in `r4` may either correspond to a byte of the lookup table used in the AES implementation ($S$-Box or $T$-table) or to a byte of the unrolled key.

```
 1. ldr r4,[r7];
 2. mov r12,r4;
 3. asr r0,r4,#1;
 4. eor r12,r12,r0;
 5. asr r0,r4,#2;
 6. eor r12,r12,r0;
 7. asr r0,r4,#3;
 8. eor r12,r12,r0;
 9. asr r0,r4,#4;
10. eor r12,r12,r0;
```

```
11. asr r0,r4,#5;
12. eor r12,r12,r0;
13. asr r0,r4,#6;
14. eor r12,r12,r0;
15. asr r0,r4,#7;
16. eor r12,r12,r0;
17. and r12,r12,#1;
18. ldr r6,[r7];
19. cmp r12,r6;
20. bne <error>;
```

The code computes the parity bit through `xor`ing a value obtained from a shifted copy of the one whose parity must

be checked. The parity value is accumulated in a scratch pad register (`r12`) and employs a temporary register (`r0`) to store the correctly shifted copy of the value whose parity must be computed. After all the 8 bits of the byte are added together, the final value is masked with a single bit mask and compared with the correct parity which is loaded from the memory in instruction 19.

Since the ARM architecture has a barrel shifter capable of shifting/rotating one of the two operands of an arithmetical-logical instruction, it is possible to skip altogether the use of the temporary register and considerably reduce the number of instructions needed to compute the parity as shown in the following code:

```
 1. ldr r4,[r7];
 2. mov r12,r4;
 3. eor r12,r4,r4, LSR #1;
 4. eor r12,r12,r4,LSR #2;
 5. eor r12,r12,r4,LSR #3;
 6. eor r12,r12,r4,LSR #4;
 7. eor r12,r12,r4,LSR #5;
 8. eor r12,r12,r4,LSR #6;
 9. eor r12,r12,r4,LSR #7;
10. and r12,r12,#1;
11. ldr r6,[r7];
12. cmp r12,r6;
13. bne <error>;
```

As far as fault coverage goes, parity codes are able to detect half of all the possible faulty results of the operation they are protecting: in particular, all single bit faults are correctly detected, but no means of correcting them is available. This implies that any multi-bit fault injected, having an even number of bit flips, will not be detected by the parity scheme, i.e. one generic multi-bit fault every two. A further possibility for an attacker to bypass the protection scheme is to inject a fault both during the loading operation and during either the branch instruction after the comparison or the loading of the correct parity value.

## 4.4 Fault coverage summary and effects of the caches

Table 1 provides a brief summary of the fault detection capabilities and fault coverage of the aforementioned methodologies.

A relevant figure emerging from the table is that all the countermeasures providing full coverage from single faults, also require a higher temporal precision from a potential attacker in order to be thwarted. In particular, the second fault should be injected into a specific clock cycle, which is only 4-11 cycles apart from the first fault. This tight timeframe is a very difficult to be matched with the current fault induction techniques which require a non negligible amount of time to reset the fault inducing mean (laser, EM or clock glitcher). Since this reload time exceeds the required 11 clock cycles by far, even for implementations running at very limited clock rates, the proposed countermeasures may be regarded as safely stopping the injection of faults for all the present implementations.

Many high-end ARM based embedded systems also integrate one or two levels of memory caches in their architecture. This architectural feature has a direct impact on the fault injection countermeasures, since it implies that the result of an operation may be reused by the CPU if the wrong computed or loaded value is held in cache.

This side effect is particularly interesting in case load instructions are replicated since inducing a fault in a single load implies in turn that all the subsequent ones will be employing the same faulty value held in the data cache. Such a side effects could allow an attacker to bypass load replication countermeasures since all the comparison made to detect the error would act on the same faulty value, thus failing to detect the error. A possible solution is the use of per-line cache invalidation, which is available on the ARM architecture through the MCR instruction. Through selective invalidation of the cache lines containing the values which have just been loaded, it is possible to avoid the fault storing effect of the caches, thus performing reliable instruction duplication.

Another advantage of flushing the cache lines containing the sensitive values is the intrinsic protection against the timing attacks mentioned in [13], since it is no longer possible to make any inference on the position of the values loaded with respect to the loading times within the algorithm.

The invalidation of cache lines will take additional instructions on cache endowed systems, which must be placed right before every sensitive instruction in the code, in order to warrant a fresh load from main memory. Thus, all the aforementioned listings will gain a number of MCR instructions equal to the number of LDR performed, since the whole inner state of the AES cipher is kept in the registers, which in turn implies that there are no memory writeback operations which need to be taken care of. It is important to notice that also the common DMR/TMR applied at algorithm level will need to flush the caches among the repeated executions of the algorithm, otherwise they would suffer from the same problems as the proposed methodologies.

In order to minimize the impact of the cache line flushes, it is possibile, if the source code of the algorithm is available, to align all the sensitive variables to the cache lines through employing compiler directives such as the DCACHEALIGN of GCC. These directives, provided the cache line size is known, allocate the variables so that their beginning in memory is aligned to a cache line, thus resulting in minimal trashing of unrelated values when a cache line is flushed. Employing this technique avoids the possible performance degradation on the remainder of the running programs on the chip which would ensue if the cache lines contained values used outside of the encryption algorithm.

The following section will present, together with the figures of cost for the algorithms in both cache endowed and cache free environments in order to provide results suitable for the whole range of embedded systems based on ARM CPUs.

# 5. COUNTERMEASURES EVALUATION

In this section we discuss the efficiency of the proposed countermeasures when applied to either the full cipher or to a selected subset of the weak spots according to the known attacks. The findings in [2, 6–10, 12] suggest that injecting a fault within the last three rounds of the AES cipher leads to successful attacks allowing a complete key recovery with as few as 6 faults for AES-256. It is thus mandatory to protect the last three rounds in full to prevent these attacks from

| Technique | Single fault coverage [%] | Faults needed to skip | Maximum fault distance for double faults |
|---|---|---|---|
| DMR | 100 | 2 | 4 |
| TMR | 100 | 2 | 11 |
| Parity | 50 | 1 | 12 |

**Table 1: Fault coverage and minimum required faults to subvert a countermeasure**

succeeding. The attack in [4] addresses another fault injection technique capable of discovering the full key through injecting single bit errors during the first key addition. This implies that also the first key load and addition must be protected against faults. Further care must be taken in protecting against the attack mentioned in [4] since, being a safe-error attack, it requires the countermeasure designer to provide an error correction mechanism in order to produce a correct result regardless of the fault injected during the first key addition. If the AES implementation outputs either a randomized value, the actual faulty ciphertext or simply signals an error without outputting anything, the attack proposed in [4] will still be able to extract informations since it only relies on detecting an anomaly in the correct functioning of the circuit. Since the fault injection capabilities required to lead the aforementioned attack are very high, it is reasonable to take into account an error correction mechanism only if the implementation is protecting significantly valuable goods.

A key point of employing a parity bit for protecting the encryption is that it cannot detect errors that were inserted during computations, thus leaving a possible target for attackers. However, a low cost and easy to setup technique described in [3] is only able to inject faults in the load operations in an ARM9 architecture and thus would be detected also by a parity bit check.

An obvious alternative countermeasure is to replicate the entire computation of the cipher and compare only the final results. While this may sound like a sufficient protection, an attacker capable of injecting faults with accurate timing may be able to produce two identically faulty results and thus bypass the check. This is due to the fact that replicating a whole algorithm execution leaves a large time gap between the two fault injection points, thus allowing the attacker to properly reload the fault injection equipment. By contrast, this is much harder, if not impossible, if the time gap is only a couple of instructions wide, as in the proposed countermeasures.

The implementation of AES used to validate our proposed countermeasures is a $T$-table based implementation, realized in `C` and compiled for the ARM9 architecture, employing release grade optimizations (`-O2`) with GCC 4.0.2. Since the ARM architecture provides free rotations through the barrel shift unit, the most efficient implementation is the one employing only a single $T$-table and rotating on the fly the obtained value to get the correct 32-bit word to update the state of the cipher, as described in Section **??**.
The compiled object was subsequently disassembled and the countermeasures were introduced directly into the assembly listing. Table 2 presents the overhead, expressed in number of clock cycles, needed for each countermeasure to protect a single instruction. The overhead has been split into individual components, namely, the extra computational instructions inserted, the additional loads and the number of scratch registers required.

**Table 2: Countermeasures overhead per single instruction to be protected**

| Countermeasure | Instruction Count | `load` Count | No. of extra Registers |
|---|---|---|---|
| **None** | 1 | 1 | 0 |
| **DMR** | 4 | 2 | 1 |
| **TMR** | 14 | 3 | 3 |
| **PAR** | 20 | 2 | 3 |
| **PAR-barrel** | 13 | 2 | 2 |

The results show that employing a TMR scheme has almost the same computational cost of calculating the byte parity (employing a tailored algorithm for the given ARM architecture) and is less expensive than the basic parity code method. Figure 1 depicts the timing overhead of protecting

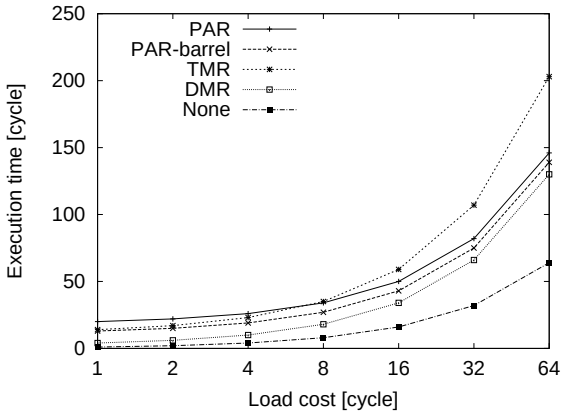


**Figure 1: Time overhead required to protect a single `load` instruction as a function of the clock cycles needed to execute it**

a single load instruction when taking into account also the cost in terms of clock cycles of the load operation. The costs range from an ideal of a single cycle (which may happen in case the value is held tightly coupled memories, typical of small embedded systems) to 64 cycles for slow off-chip memory.

From the figure one can notice that the DMR is uniformly less expensive than the parity schemes regardless of the latency of the memory, while providing the same error detection capability. The TMR has a performance overhead lower than that of the parity scheme up to memory latencies of 8 cycles, thus being a viable solution when either fast on-chip memories are employed or if on-the-fly error correction is a stringent requirement.

Figures **??** and **??** depict the overhead in number of cycles for the AES algorithm when applying complete protection

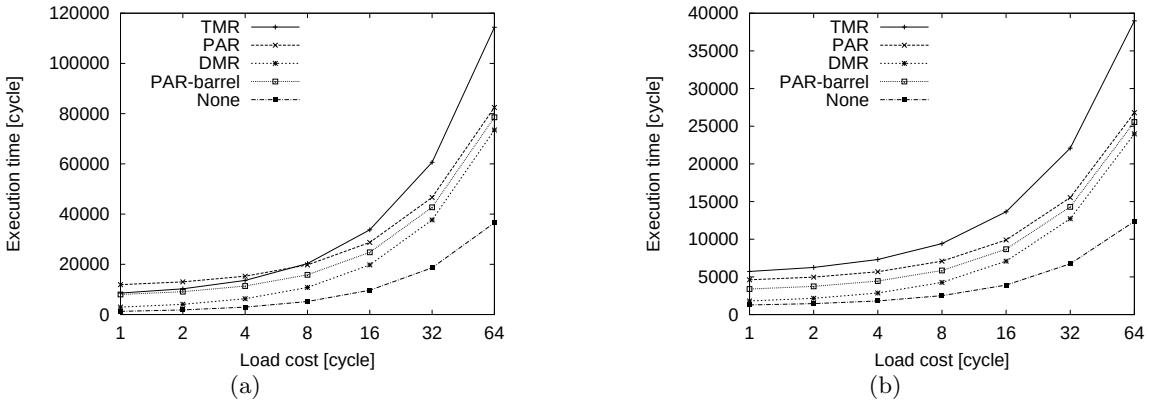


**Figure 3: Execution times of AES with protection of all `load` instructions (a), and with protection of the `load` instructions in the last three rounds only (b)**
without cache flushing

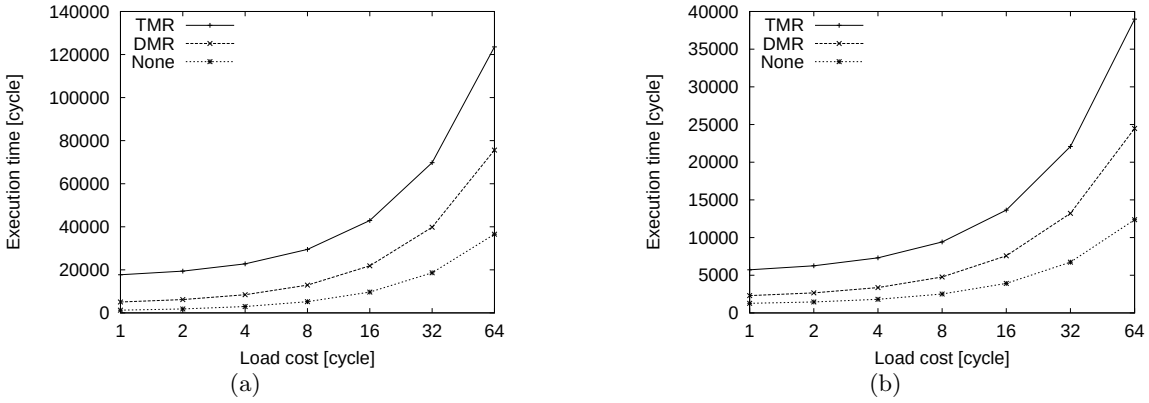


**Figure 4: Execution times of AES with protection of all the instructions(a), and with protection of the `load` instructions in the last three rounds only (b)**
without cache flushing

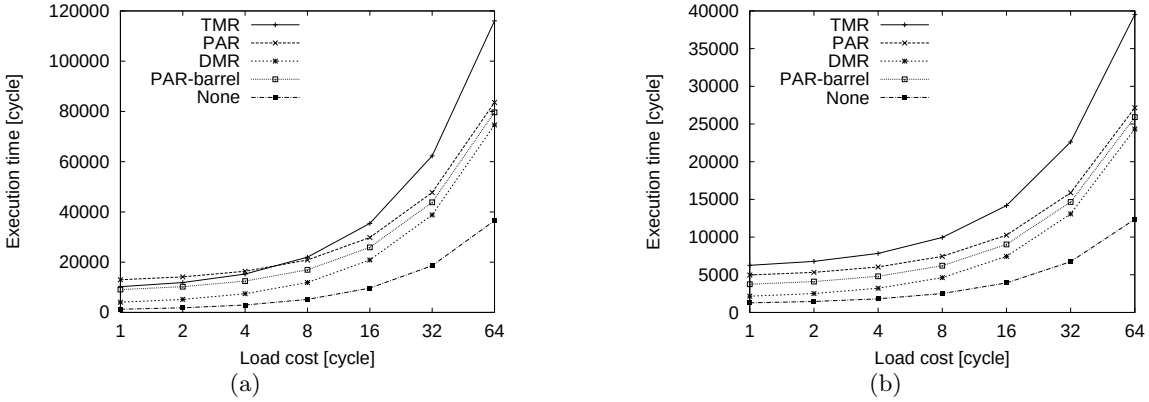


**Figure 5: Execution times of AES with protection of all `load` instructions (a), and with protection of the `load` instructions in the last three rounds only (b)**
with cache flushing

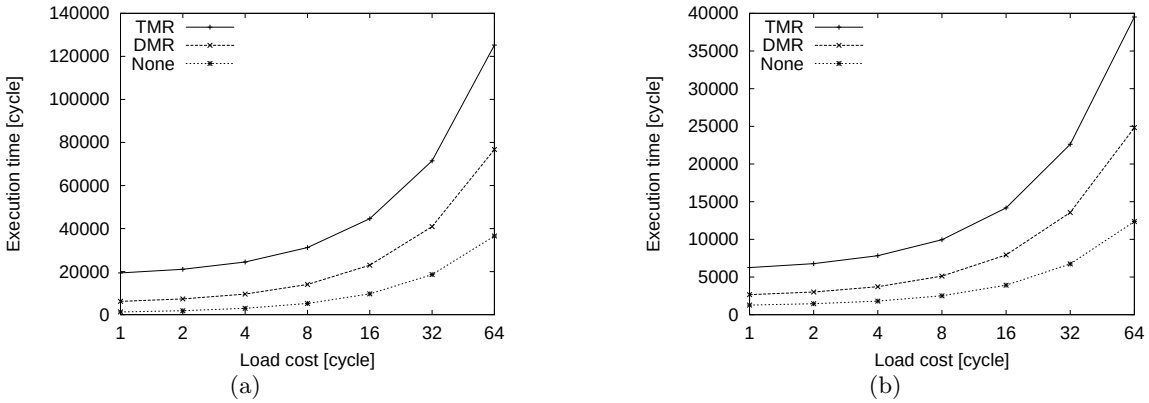


**Figure 6: Execution times of AES with protection of all the instructions(a), and with protection of the `load` instructions in the last three rounds only (b)**
with cache flushing

**Table 3: Overview of clock cycles overhead for all the proposed countermeasures schemes**

| **Countermeasure** | **Last Three Round Protection** | | **Whole Algorithm** | |
|---|---|---|---|---|
| | `load` cycles: 2 | `load` cycles: 64 | `load` cycles: 2 | `load` cycles: 64 |
| **None** | 1 | 1 | 1 | 1 |
| **DMR (`load` instr.)** | ×1.48 | ×1.94 | ×2.23 | ×2.01 |
| **TMR (`load` instr.)** | ×2.82 | ×2.98 | ×5.60 | ×3.13 |
| **PAR (`load` instr.)** | ×3.54 | ×2.18 | ×7.13 | ×2.26 |
| **PAR-barrel (`load` instr.)** | ×2.45 | ×2.05 | ×4.99 | ×2.15 |
| **DMR (full)** | ×1.83 | ×1.98 | ×3.39 | ×2.07 |
| **TMR (full)** | ×4.30 | ×3.15 | ×10.63 | ×3.38 |

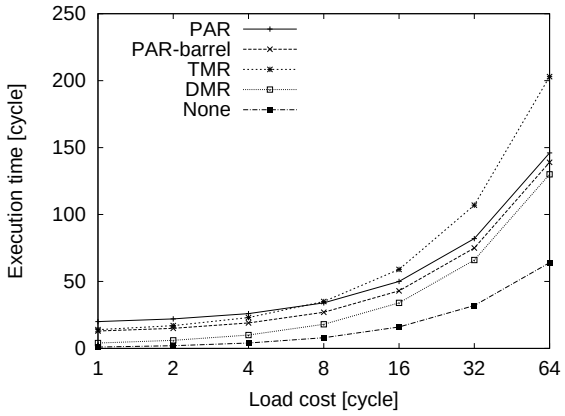


**Figure 2: Time overhead required to protect a single `load` instruction as a function of the clock cycles needed to execute it including cache flushing**

(to all the rounds) or when selectively protecting only the sensitive spots mentioned above (last three rounds plus the initial key addition). For this comparison, the countermeasures are limited to the `load` instructions. The results in these two figures confirm the intuition deduced from the single `load` overhead investigation, suggesting DMR as the cheapest fault detection scheme and TMR as a reasonably lightweight alternative if error correction is desired. The comparison between the blind injection of the countermeasure (Figure **??**) and the selective protection of the sensitive spots (Figure **??**) shows an advantage of roughly three times in terms of clock cycle overhead for the selective solution. Table 3 provides a full overview of the countermeasures proposed in this paper together with their time overheads. The comparison among the methods to protect only the load instructions shows that proper selective duplication (DMR) is cheaper than all the other methods, and clearly cheaper than duplicating the entire cipher. This scheme also provides a stronger protection than the simple recomputation since it can only be eluded by an attacker able to inject two precisely timed and equal faults in two instants only a single clock cycle apart: this is expected to be more difficult than injecting two identical faults into two subsequent runs of the same algorithm to circumvent simple doubled execution.

The TMR scheme, applied only to the sensitive parts, results in a reasonably lightweight scheme, keeping the overhead lower than the trivial triple execution of the algorithm and outperforming also the straightforward parity scheme (which does not provide error correction) if the memory is fast enough. This suggests that TMR is a viable alternative when error correction is desired or when an even tighter bound on the attacker capabilities must be placed. In fact, the only way to circumvent the per-load-instruction TMR is to inject three identical faults in three subsequent instructions, which has not yet proved feasible due to the difficulty of the fault injection apparatus in disturbing a circuit sufficiently quickly and precisely.

The last two rows in Table 3 present the overheads of protecting all the instructions using DMR and TMR, respectively. Albeit at a higher cost, these schemes provide complete protection against any possible injected fault, either in the computational part or in the memory accesses. Particularly interesting are the results obtained for the last three round protection when DMR is employed: the time overheads are lower than those for a trivial duplication of the whole computation, while retaining per-instruction consistency checking. Another noteworthy result is for the case when TMR is applied to the whole algorithm on devices with slow memories. In this case, with an overhead of only 10% higher than a triplication of the entire algorithm, it is possible to provide instruction level triplication, checking and correction, thus providing a complete protection against all known fault attacks using all the currently known fault injection techniques.

# 6. CONCLUSION

In this paper we have explored possible countermeasures against faults attacks on software implementations of AES that are based on introducing redundant computations. The proposed countermeasures (selective, per-instruction, DMR and TMR) require lower overheads than common alternatives (parity bit checking) and can be applied in an automated fashion to a software implementation of the AES algorithm.

We foresee possible future development in evaluating the effectiveness of the proposed countermeasures for other architectures and ciphers.

# 7. REFERENCES

[1] R. J. Anderson and M. G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, 1998. Springer-Verlag.

[2] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. Low Voltage Fault Attacks to AES. In M. Tehranipoor and J. Plusquellic, editors, *HOST*, pages 7–12. IEEE Computer Society, 2010.

[3] A. Barenghi, G. M. Bertoni, E. Parrinello, and G. Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, volume 0, pages 23–31, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[4] J. Blömer and J.-P. Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In R. N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2003.

[5] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[6] P. Dusart, G. Letourneux, and O. Vivolo. Differential Fault Analysis on A.E.S. *CoRR*, cs.CR/0301020, 2003.

[7] C. Giraud. DFA on AES. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004.

[8] A. Moradi, M. T. M. Shalmani, and M. Salmasizadeh. A Generalized Method of Differential Fault Attack Against AES Cryptosystem. In L. Goubin and M. Matsui, editors, *CHES*, volume 4249 of *LNCS*, pages 91–100. Springer, 2006.

[9] G. Piret and J.-J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with

Application to the AES and KHAZAD. In C. D. Walter, Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[10] N. Selmane, S. Guilley, and J.-L. Danger. Practical Setup Time Violation Attacks on AES. In *EDCC-7'08: Proceedings of the 2008 Seventh European Dependable Computing Conference*, pages 91–96, Washington, DC, USA, 2008. IEEE CS.

[11] S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[12] J. Takahashi and T. Fukunaga. Differential Fault Analysis on AES with 192 and 256-Bit Keys. Cryptology ePrint Archive, Report 2010/023, 2010. `http://eprint.iacr.org/`.

[13] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.