Buying AES Design Resistance with Speed and Energy

Jean-Michel Cioranesco[1], Roman Korkikian[1], David Naccache[1,2], Rodrigo Portella do Canto[1]

[1] Université Panthéon Assas
12 Place du Panthéon, 75005, Paris, France.
jean-michel.cioranesco@etudiants.u-paris2.fr
roman.korkikian@etudiants.u-paris2.fr
rodrigo.portella-do-canto@etudiants.u-paris2.fr

[2] École normale supérieure
Département d'informatique
45, rue d'Ulm, 75005, Paris CEDEX 05, France.
david.naccache@ens.fr

Keywords: AES, power scrambling, power attack, CPA, fault attack, half size memory.

Abstract. Fault and power attacks are two common ways of extracting secrets from tamper-resistant chips. Although several protections have been proposed to thwart these attacks, resistant designs usually claim significant area or speed overheads. Furthermore, circuit-level countermeasures are usually not reconfigurable at runtime. This paper exploits the AES' algorithmic features to propose low-cost and low-latency protections. We provide Verilog and FPGA implementation details. Using our design, real-life applications can be configured during runtime to meet the user's needs and the system's constraints.

1 Introduction

The Advanced Encryption Standard (AES) algorithm, also known as Rijndael, is a widely used block-cipher standardized by NIST in 2001 [1]. Compared with its predecessor DES [1,2], the AES features longer keys, larger plaintexts and more involved basic binary transformations [3].

Despite the fact that AES is mathematically safer than the DES, straightforward AES *implementations* are not necessarily secure and several authors [4,5,6] have exhibited ways of exploring information that leaks from AES implementations. Such leakage is typically power consumption, electromagnetic emanations or the time required to process data. Additional constraints such as fault-resistance, chip technology, performance, area, power consumption, and even patent compliance further complicate the design of real-life AES coprocessors.

This article addresses resistance against two physical threats: power and fault attacks. The proposed AES architecture leverages the algorithm's structure to create low-cost protections against these attacks. This allows very flexible runtime configurability without significantly affecting performance.

The remaining of the paper is organized as follows: Section 2 recalls the AES' main features and proposes an architecture for implementing it. Section 3 explains how to add power scrambling and fault detection to the proposed implementation. The result is a chip design allowing 29 different software-controlled runtime configurations. Section 4 introduces an idea of reducing the memory required to store *state keys* in the decryption mode. Section 5 compares simulation and synthesis results between an unprotected AES and our protected implementations. While Section 6 concludes this article, Section 7 proposes further research about a novel type of attack.

2 The Proposed AES Design

The AES is a symmetric iterative block-cipher that processes 128-bit blocks and supports keys of 128, 192 or 256 bits [1]. Key length is denoted by $N_k = 4, 6,$ or 8, and reflects the number of 32-bit words in the key. At start, the 128-bit plaintext P is split into a 4×4 matrix S of 16 bytes called *state*. The *state* goes through a number of rounds to become the ciphertext C.

The number of rounds N_r is a function of N_k. Possible $\{N_r, N_k\}$ combinations are $\{10, 4\}$, $\{12, 6\}$ and $\{14, 8\}$. A particular round $1 \leq r \leq N_r$ takes as input a 128-bit *state* $S^{[r]}$ and a 128-bit *round key* $K^{[r]}$ and outputs a 128-bit *state* $S^{[r+1]}$. This is done by successively applying four transformations called *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*.

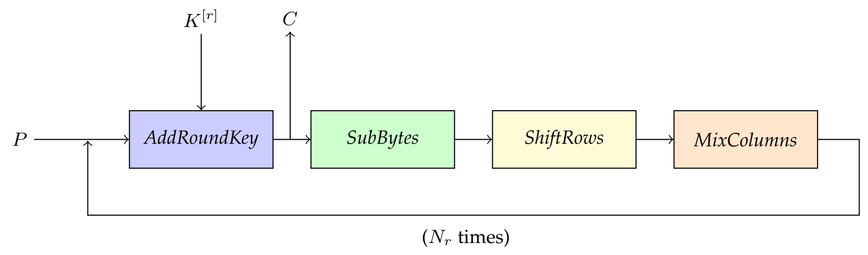

Fig. 1. AES Encryption Flowchart.

AES encryption starts with an initial *AddRoundKey* transformation followed by N_r rounds consisting of four transformations, in the following order: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. *MixColumns* is skipped in the final round ($r = N_r$). If during the last round *MixColumns* is bypassed, we can look upon the AES as the 4-block iterative structure shown in Fig.1.

Decryption has a similar structure in which the order of transforms is reversed (Fig. 2) and where inverse transformations are used (Note that *AddRoundKey* is idempotent). In both designs, a register barrier at the end of each transformation block is used to save intermediate results. Therefore the intermediate information that eventually yields $S^{[r]}$ is saved four times during each AES round. It takes $4N_r + 1$ clock cycles to encrypt (or decrypt) a data block using this design.

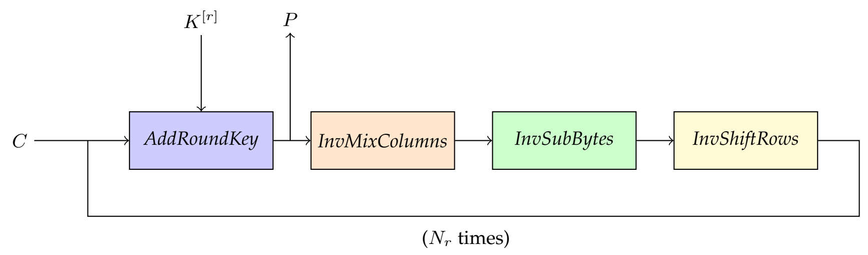

Fig. 2. AES Decryption Flowchart.

Fig. 1 and Fig. 2 show that, during each clock cycle, only one block of the chain actually computes the *state*, while the other three blocks are processing useless data. This is potentially risky, as the three concerned blocks "chew" computationally useless data related to P (or C) and $K^{[r]}$ and thereby expose the design to unnecessary side-channel attacks.[1] This computation is shown in Fig. 3 where red arrows represent the path of usefully active combinatorial logic.

3 Energy and Security

3.1 Power Analysis

We assume that the reader is familiar with the power [6] and fault [7] attacks that we do not remind here.

To benchmark our design the AES was implemented on FPGA. Power was measured at 1GS/s sampling rate with 250MHz bandwidth using PicoScope 3407A oscilloscope. To guarantee the identical conditions every new plaintext was given to the FPGA at the same clock after the reset.

We performed a Correlation Power Attack (CPA) on the first AES Sbox output since Sbox operation is generally considered as the most power gluttonous. Our

[1] In that respect see our open question in Section 7.

power model was based on the number of flipped register's bits in the Sbox module when the initial register's barrier R_0 is rewritten with the Sbox output as follows:

$$\mathrm{HD}(Sbox[P \oplus K_0], R_0) = \mathrm{HW}(Sbox[P \oplus K_0] \oplus R_0) \tag{1}$$

where R_0 is the previous register's state; P is a given plaintext; K_0 is the AES master key.

The value R_0 was assumed to be constant since all the encryptions were performed at the same clock after the reset. When R_0 could not be computed then all possible 256 values were tried. Pearson correlation coefficient was used to link the model and the genuine consumed power.

The following section presents a reference evaluation of the unprotected AES implementation showing its vulnerability compared to two (LFSR and tri-state buffers) side-channel countermeasures introduced later.

3.2 Power Scrambling

It is a natural idea to shut down unnecessarily active blocks. To do so, each block receives a new 1-bit input named *ready* activating the block when *ready* $= 1$. If *ready* $= 0$, the block's pull-up resistors are disconnected using a tri-state buffer connected to the power source. This saves power and also prevents the circuit from leaking "unnecessary" side-channel information.

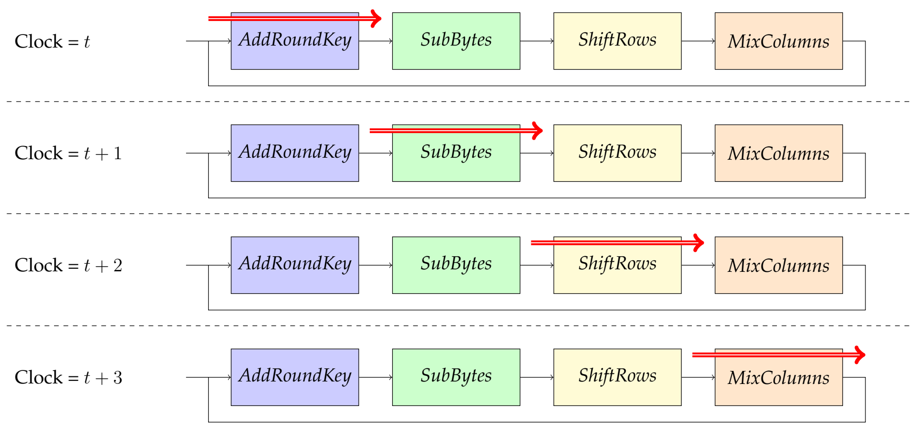

Fig. 3. Flow of Computation in Time.

Logically the pipeline architecture that we have just described has to be less vulnerable against First Order DPA attacks. Its four register barriers introduce additional noise, so we expect that the correlation shall be at least smaller that for the AES design with one round per clock computation.

To asses the security of each proposed design, we will compare an incorrect key byte correlation to a correct key byte correlation. Fig. 4 shows these two coefficients. As expected, the correct key is correlated to the power traces, however even for 500,000 traces Pearson correlation coefficient is smaller than 0.015. Anyway, this implementation is vulnerable.

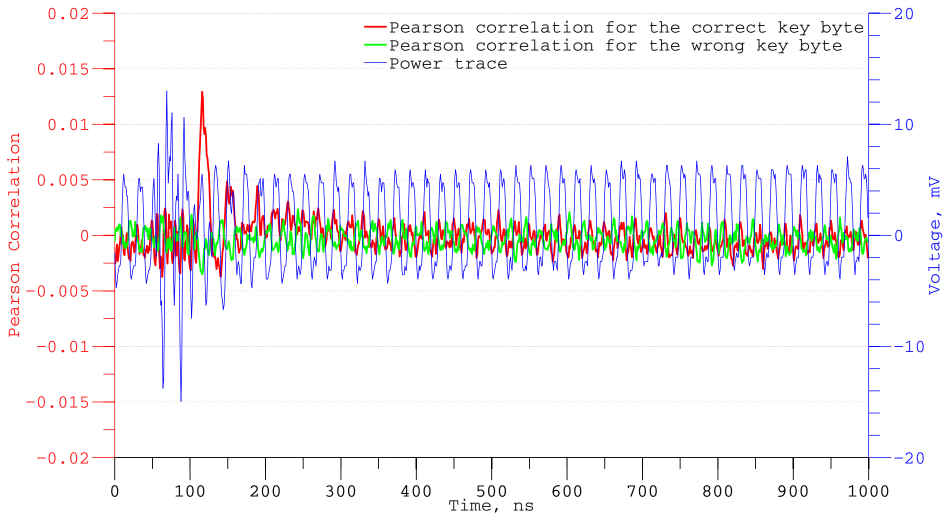

Fig. 4. Unprotected implementation: Pearson correlation value of a correct (red) and an incorrect (green) key byte guess. 500,000 power traces.

To exploit the unused blocks to hide the device's power signature even better we propose two modifications. The first consists in injecting (pseudo) random data into the unused blocks, making them process that random data. Subsequently, three of the four blocks will consume power in an unpredictable manner. Note that because we use the exact same gates to compute and to generate noise, the expected spectral and amplitude characteristics of the generated noise should mask leakage quite well. Although any random generator may be used as a noise source, we performed our experiments using a 128-bit LFSR. An LFSR is purely coded in digital HDL, making tests easier to implement.

Fig. 5 shows that a multiplexer controlled by the *ready* signal selects either the useful intermediate *state* information or the pseudo-random LFSR output. For the *AddRoundKey* block, LFSR data replaces the key. Therefore when *AddRoundKey*'s *ready* $= 0$, pseudo-random data (unrelated to the key) are xored with the *state* coming from the previous block (*MixColumns* if encrypting, *InvShiftRows*

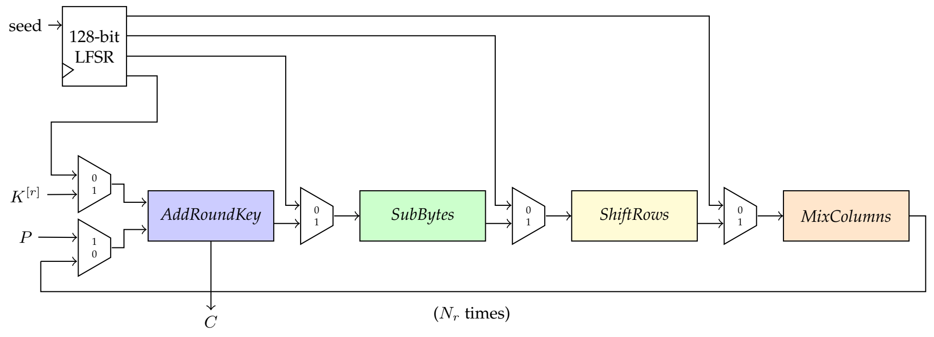

Fig. 5. Power Scrambling with a PRNG.

if decrypting). For the other blocks, the pseudo-random data replaces the *state* when $ready = 0$.

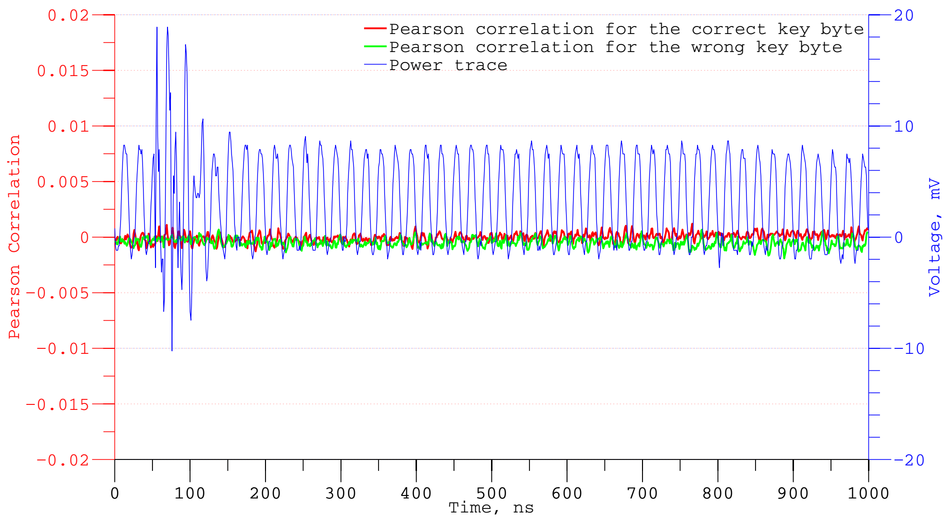

Fig. 6. LFSR implementation: Pearson correlation value of a correct (red) and an incorrect (green) key byte guess. 1,200,000 power traces.

Attacks performed on this implementation revealed that this countermeasure increases key lifetime. Fig. 6 is the equivalent of Fig. 4 for the protected implementation using an LFSR. The correct key correlation can not be distinguished from the incorrect key correlation even with 1,200,000 traces. However, we assume that this implementation still might be vulnerable if more traces are acquired or if Second Order DPA is applied.

Real-life implementations must use true random generators. Indeed, if a deterministic PRNG seed is used the noise component in all encryptions becomes constant and cancels-out when computing differential power curves.

A second design option interleaves tri-state buffers between blocks to hide power consumption. By shutting down the three useless blocks, we create a scrambled power trace where one block computes meaningful data while the other three "process" high impedance inputs, which means that these blocks "compute" leakage current coming from their inputs.

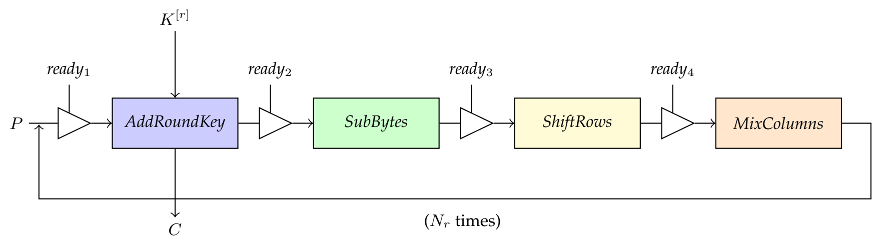

Fig. 7. Power Scrambling with Tri-State Buffers.

As illustrated in Fig. 7, the input signal *ready*$_i$ determines which blocks are tri-stated and which block is computing the AES *state*. In other words, the *ready*$_i$ signal "jumps" from one block to the next, so that only one block is computing while the other three are scrambling the power consumption. Although this solution has a smaller overhead in terms of area (as it does not require random number generation) tri-state buffers tend to be slow. Furthermore, the target environment (FPGA or IC digital library) must offer tri-state cells.

The experimental results we obtained on FPGA were surprising, we couldn't attack the design with 800,000 power traces. The correlations shown in Fig. 8 do not allow to visually distinguish the correct key from a wrong guess. As before we assume that this implementation can be still attackable if more power traces are acquired or if Second Order DPA is applied.

A full study of this solution would require an ASIC implementation with real tri-state buffers, as an FPGA emulates these buffers and may turn out to be resistant because of an undesired CLB mapping side effects.

3.3 Transient Fault Detection

We will now use idle blocks to check for transient faults. Each block in the chain can "stutter" during two consecutive clock cycles to recompute and check its own calculation. For instance, as shown in Fig. 9, at clock t, a given block B_i receives a *ready*$_i$ signal, computes the *state* and saves it in the register barrier R_i.

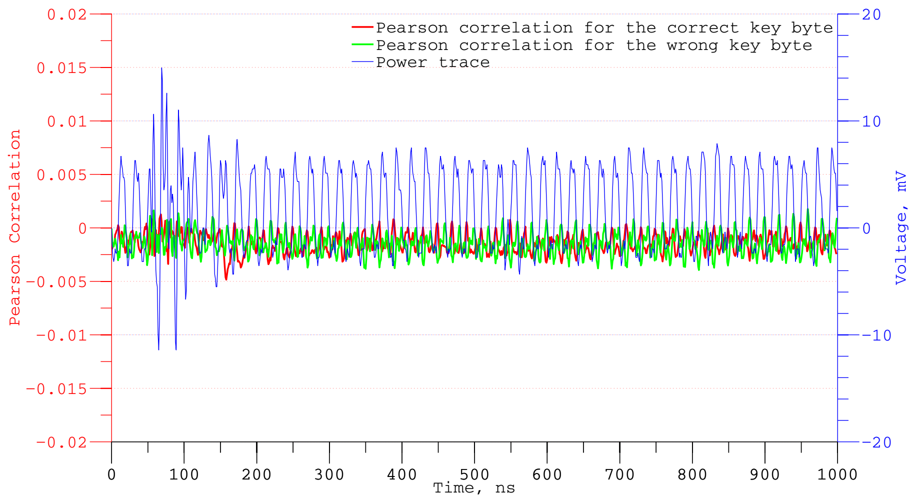

Fig. 8. Tri-state buffers implementation: Pearson correlation value of the correct key byte (green) and a wrong key byte guess (red). 800,000 power traces.

At clock $t + 1$, the result enters the next block $B_{i+1 \bmod 4}$ which is now working, while B_i reverts to checking, *i.e.*, B_i recomputes the same output as at clock t and compares it to the saved B_i value. This process is repeated for the other blocks in the chain. If any transient fault happens to cause a wrong result at the output of any block, the error will be detected within one clock cycle.

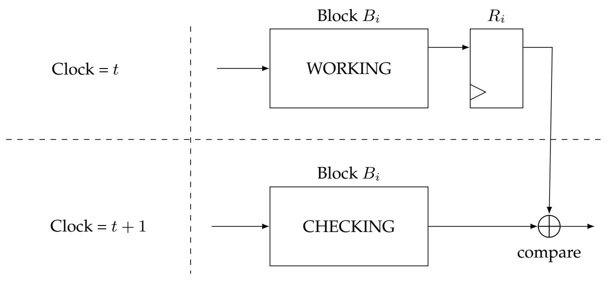

Fig. 9. Transient Fault Detection Scheme for AES.

3.4 Permanent Fault Detection

The AES structure of Section 2 also allows us to use one block of the chain to compute a pre-determined plaintext or ciphertext. The encryption (or decryption) of a chosen input (*e.g.* the all-zero input Z) is pre-computed once for all and hardwired (let $W = \text{AES}(Z)$ denote this value). While the system processes the actual input through one block (out of four) during any given clock cycle, another block is dedicated to recompute W. One clock after the actual C emerges, AES(Z) can be compared to the hardwired reference value W. If $W \neq$ AES(Z), a transient or a permanent fault occurred.

In this scenario, the system starts by computing AES(Z) in the first clock cycle, followed by the actual computation of C. This allows the implementation to check up all the blocks during the execution and make sure that no permanent fault occurred. In the last clock cycle, while C is being processed in the last block, the correctness of AES(Z) is compared with the hardwired value before outputting C.

In Fig. 10, the red arrows represent data flow through the transformation blocks. After the initial clock cycle, the first block starts computing C. The WORKING blocks represent the calculation of C. The CHECKING blocks represent the calculation of AES(Z).

While AES(Z) will be calculated in $4N_r + 1$ clock cycles, C will be calculated in $4N_r + 2$ cycles. If the fault needs to be caught earlier, the solution described in [8] can be adapted. Yet another option consists in comparing intermediate Z encryption results (*i.e.* intermediate *state* values) to hardwired ones. Note that our design differs from [8] where a the decryption block is used for checking the encryption's correctness [3].

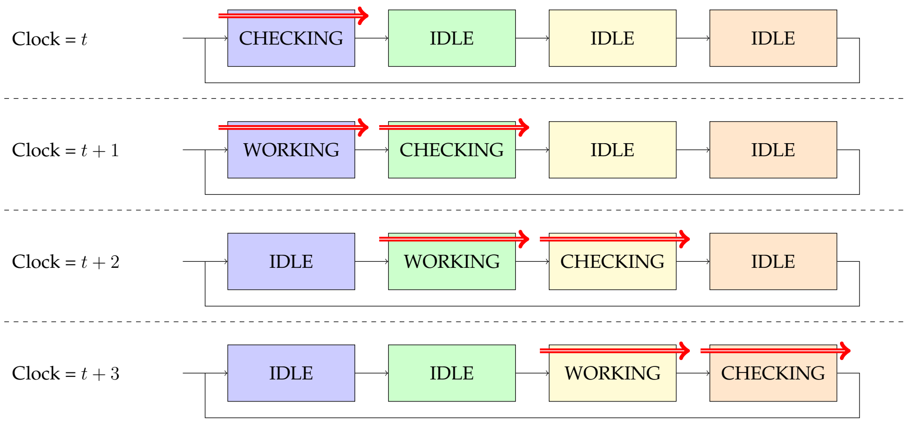

Fig. 10. Permanent Fault Detection Scheme for AES.

3.5 Runtime Configurability

The proposed AES architecture is a 4-stage pipeline where each stage can be used independently of the others. As already noted, blocks can perform five different tasks:

- Compute a meaningful state;
- Be in idle state to save energy;
- Scramble power consumption;
- Check for transient faults by recomputing previous calculation;
- Check for permanent faults by computing a known input.

To explore all possible combinations, we proceed as follows: first, we generate all $5^4 = 625$ combinations (5 operations for 4 transformation blocks). We can consider a subset of these combinations if we work with 4 operations only, and remember that each E entry represents two actual options (tri-state or idle). This reduces the number of combinations to $4^4 = 256$. We eliminate all configurations that are circular permutations of others, *i.e.* already counted configurations shifted in time. We also eliminate the meaningless configurations in which there isn't at least one block computing. All configurations having more than one permanent fault protection block at a time are removed as they don't add any extra protection. Finally, we eliminate the cases where a transient fault checking is not preceded by a computing block or by a permanent fault verification.

Table 1. 29 Possible Configurations.

	Block 1	Block 2	Block 3	Block 4
	C	C	C	C
	C	C	C	E
	C	C	C	T
	C	C	C	P
	C	C	E	E
	C	C	E	T
	C	C	E	P
	C	C	T	T
	C	C	T	P
	C	C	P	E
	C	C	P	T
	C	E	C	E
	C	E	C	T
	C	E	C	P
	C	E	E	E
	C	E	E	T
	C	E	E	P
	C	E	T	T
⋆	C	E	T	P
	C	E	P	E
⋆	C	E	P	T
	C	T	C	P
	C	T	T	T
	C	T	T	P
⋆	C	T	P	E
	C	T	P	T
	C	P	E	E
⋆	C	P	E	T
	C	P	T	T

Table 2. Number of Configurations.

C	E	P	T	Configurations
4				1
3	1			1
1	3			1
3		1		1
3			1	1
1			3	1
2			2	1
1	1		2	1
1		2	1	1
2	2			2
1		1	2	2
2	1	1		3
1	2	1		3
1		1	2	3
2		1	1	3
1	1	1	1	4

Table 1 shows that the design can perform 29 different task combinations, where *C* stands for computing, *E* stands for energy (power scrambling, idleness or any combination of these two if there are more than two *Es* in the considered configuration), *T* stands for transient fault checking and *P* stands for permanent fault checking. These options can be activated *during runtime* according to the system's constraints such as power consumption or speed. If there are no specific requirements, we recommend any of the four best configurations protecting against all attacks at once. These are singled-out in Table 1 by a $\star$.

Table 2 shows the number of configurations per protection goal. Note that for a given protection goal, different configurations can be alternated between executions without any performance loss.

4 Halving the Memory Required for AES Decryption

As we have seen, it takes $4N_r+1$ clock cycles to encrypt or decrypt an input. The first block of the chain, *AddRoundKey* xors the *state* with the *subkey*. Therefore, the *key expansion* block is designed to deliver a new 32-bit *subkey* chunk at each clock cycle.

When decrypting, the AES uses *subkeys* in the reverse order, so all *subkeys* need to be expanded and stored in memory before decryption starts. For that, decryption requires a $128N_r$-bit buffer. These $128N_r$ bits are stored in a register having N_r records of 128 bit each. Nevertheless, it is possible to halve the number of records by using the following idea: let sk_{N_r} be the *subkey* required at round N_r. All *subkeys* are computed but only the last $N_r/2$ *subkeys* are stored in memory. After the first 4 clock cycles, *AddRoundKey* block uses sk_{N_r} (the first *AddRoundKey* uses the initial *key* sk_0 which we assume to be already recorded). After 4 more cycles, sk_1 is saved in the record previously occupied by sk_{N_r}. The buffer continues to be used in such a way that each previously used (*i.e.* read) *subkey* is replaced by a new *subkey* of rank smaller than $N_r/2$. By the time that AES decryption requires $sk_{N_r/2}$, the *subkeys* sk_1 to $sk_{N_r/2-1}$ would have already been replaced *subkeys* sk_{N_r} to $sk_{N_r/2}$.

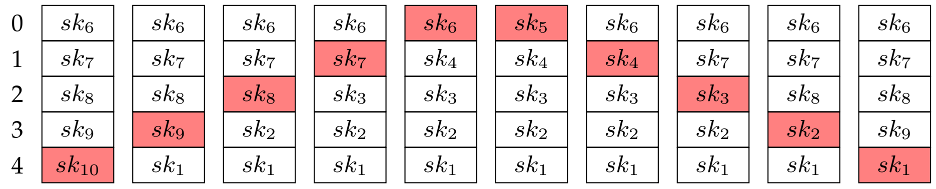
0 1 2 3 4
sk_6 sk_7 sk_8 sk_9 sk_{10}
sk_6 sk_7 sk_8 sk_9 sk_1
sk_6 sk_7 sk_8 sk_2 sk_1
sk_6 sk_7 sk_3 sk_2 sk_1
sk_6 sk_4 sk_3 sk_2 sk_1
sk_5 sk_4 sk_3 sk_2 sk_1
sk_6 sk_4 sk_3 sk_2 sk_1
sk_6 sk_7 sk_3 sk_2 sk_1
sk_6 sk_7 sk_8 sk_2 sk_1
sk_6 sk_7 sk_8 sk_9 sk_1

Fig. 11. Memory Halving for AES Decryption When $N_r = 10$.

As shown in Fig. 11, only 5 records are required when $N_r = 10$. Analogously, $\{6, 7\}$ records are required for $N_r = \{12, 14\}$. The red positions are *subkeys* being used at each *AddRoundKey* operation, from left to right. Note that we assume that the initial *key* sk_0 is known and does not need to be stored.

The algorithm is formally defined as follows: Create a buffer of $N_r/2$ records denoted $r[0], \ldots, r[N_r/2 - 1]$. Place in each $r[i]$ the *subkey* $sk_{i+1+N_r/2}$.

Define the function:

$$f(i) = \frac{|2i - N_r - 1| - 1}{2}$$

When sk_i is needed, fetch it from $r[f(i)]$. After this fetch operation update the record $r[f(i)]$ by writing into it sk_{N_r-i+1}.

5 Implementation Results

A 128-bit datapath AES encryption core was coded and tested in Verilog and compiled using Cadence *irun* tool. Cadence *RTL Compiler* was used to map the design into a 45nm *FreePDK* open cell digital library. Fig. 12 represents the inputs and outputs of the AES core. The module contains a general clock signal called *CLOCK_IN*, an asynchronous low-edge reset called *RESET_IN* and a *READY_IN* signal that flags the beginning of a new encryption. Plaintext is fed into the device *via* the 128-bit bus *TEXT_IN*, while the 128-bit key is fed to the system through the input called *KEY_IN*. The module outputs two signals: *TEXT_OUT*, which contains the resulting plaintext and *READY_OUT*, that represents a valid output.

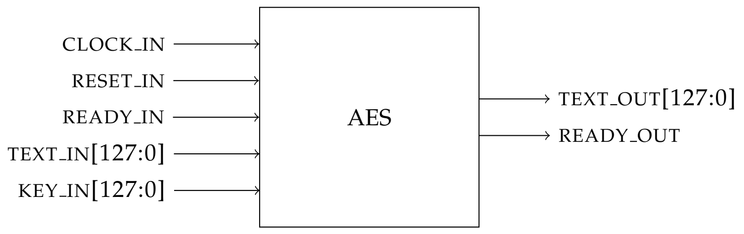

Fig. 12. AES Design's Inputs and Outputs.

Table 3 compares an unprotected AES core to the countermeasures described in this paper. The increase in terms of area is $\sim 6\%$ for the LFSR implementation and $\sim 4\%$ for the tri-state design. The LFSR implementation showed almost no increase in terms of power consumption. Since tri-state buffers shut down three out of four blocks per clock, we expect a reduction in the power consumption. The tri-state design saves roughly 20% of power compared to the unprotected AES. As tri-state buffers tend to be slower, this design lost 20% in terms of clock frequency and throughput, while the LFSR version showed no speed loss, as expected.

Table 3. Unprotected AES, LFSR and Tri-State Buffer Designs Synthesized to the 45nm *FreePDK* Open Cell Library.

	Unprotected	**LFSR**	**Tri-state**
Area (μm^2)	61,581	65,194	64,243
Number of cells	10,643	11,035	11,162
sequential	783	911	787
inverters	1,483	1,614	1,493
logic	8,375	8,506	8,368
buffers	2	4	2
tri-state buffers	0	0	512
Total power (*mW*)	2.10	2.16	1.68
leakage power	1.20	1.28	1.26
dynamic power	0.89	0.87	0.41
Timing (*ps*)	645	645	806
Frequency (*GHz*)	1.55	1.55	1.24
Throughput (*Gbit/s*)	4.84	4.84	3.87

Table 4 shows the three designs benchmarks in FPGA. They were coded in Verilog and synthesized to the Spartan3E-500 board using the Xilinx ISE 14.7 tool. LFSR and tri-state designs showed an area overhead of $\sim$ 15% compared to the unprotected AES implementation. In terms of performance, LFSR design showed no loss, while the tri-state core lost $\sim$ 7%.

Table 4. Spartan3E-500 Utilization Summary Report.

	Unprotected	**LFSR**	**Tri-state**
Number of Occupied Slices	1,994	2,290	2,296
Number of Flip Flops	1,142	1,270	1,146
Number of LUTs	3,521	4,106	4,031
Timing (*ns*)	10.789	10.714	11.580
Frequency (*MHz*)	92.68	93.33	86.35
Throughput (*Mbit/s*)	289.3	291.3	269.6

6 Conclusion

We described an unprotected AES implementation sliced in four clock cycles per round. Making use of this approach, we built on top of the unprotected core two power scrambling ideas to thwart side-channel attacks, such as CPA. We also demonstrated how the design can also prevent fault injection by re-computing its internal *state* values or by compromising one out of four blocks at each clock to compute the encryption of a known plaintext. We then exhibited simulation results and showed the comparison of the unprotected against the protected cores. The results confirm that the overhead in terms of area, power and performance is small, making this countermeasure attractive.

Moreover, the proposed AES architecture provides different options to tune the design into the user's need. Among 29 different configurations, examples include: to make the proposed AES a 4-stage pipeline (*i.e.*, compute four different plaintexts per execution), to use three blocks to generate noise against power attacks, or to use one inactive block in the chain to recompute for encryption correctness. In addition to the proposed AES implementation, we presented a simple scheme to halve the number of memory positions required for storing *subkeys* when AES is performing decryption.

7 Further Research: Ghost Data Attacks?

The footnote in Section 2 raises an interesting question: is it possible to exploit leakage from uselessly active circuit blocks to infer information about P, C or K? In this model the attacker is not allowed to access the side-channel information resulting from the actual computation of the active block (that we can assume to be ideally protected or not leaking) but only the side-channel information leaked by the three uselessly active blocks. To the best of our knowledge such attacks, that we call *ghost data attacks*, were never considered in the literature.

8 Acknowledgments

The authors thank Ms. Natacha Laniado (natacha@laniado.fr) for proofreading and correcting this paper.

References

1. National Institute of Standards and Technology (NIST), *Announcing the Advanced Encryption Standard (AES)*, November 2001.

2. M.-L. Akkar and C. Giraud, "An Implementation of DES and AES, Secure Against Some Attacks," in *CHES'01*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 309–318, Springer, 2001.
3. 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, 2003.
4. P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO'96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113, Springer, 1996.
5. P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO'99*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397, Springer, 1999.
6. S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
7. M. Joye and M. Tunstall, eds., *Fault Analysis in Cryptography*. Information Security and Cryptography, Springer, 2012.
8. G. Bertoni, L. Breveglieri, I. Koren, and V. Piuri, "Fault Detection in the Advanced Encryption Standard," in *Proc. Conf. Massively Parallel Computing Systems*, pp. 92–97, 2002.