A Grain in the Silicon: SCA-Protected AES in Less than 30 Slices

Pascal Sasdrich
Horst Görtz Institute for IT-Security
Ruhr-Universität Bochum, Germany
Email: pascal.sasdrich@rub.de

Tim Güneysu
University of Bremen & DFKI, Germany
Email: tim.gueneysu@uni-bremen.de

***Abstract*—AES is the predominant block cipher used worldwide in many cryptographic applications. Despite of the wealth of already available implementations, we here introduce an ultra-lightweight AES-128 implementation specifically tailored for reconfigurable hardware. Our basic proposal presents a full AES-128 providing 9.12 Mbit/s throughput and occupying just 21 slices of a Spartan-6 and no additional memories. We also show that this architecture almost inherently supports shuffling as side-channel countermeasure and provide results of a practical evaluation. Our protected design fits into 24 slices providing 7.82 Mbit/s throughput. Finally, we present a complete AES core that combines previous results with random number generation which fits 28 slices at 4.35 Mbit/s throughput.**

I. Introduction

The Advanced Encryption Standard (AES) is the predominant choice for standardized cryptography for virtually any device that has a need for data confidentiality and/or authentication. Since its standardization, a lot of hardware implementations have been proposed spanning over a wide range of design rationales and optimization goals, such as high-throughput, low-area, low-power or low-energy implementations. Although a high-performance cryptographic engine providing bulk data encryption is essential for many security related functions, in some cases the situation is somewhat different. Although the AES has been carefully designed to provide a high security guarantee at reasonable implementation cost in software *and* hardware, it still remains a rather expensive solution when being deployed in devices designed for lowest costs (e.g., one of the smallest, physically unprotected ASIC implementation requires 2400 GE [19]). This fact was also one of the reasons for the strong research interest in lightweight cryptography, giving rise to a wealth of other very small block ciphers, such as PRESENT [3], Noekeon [9], PRINCE [4], or SIMON [2].

Nevertheless, the situation remains unsatisfying when designers are faced with the strict requirement to integrate standardized cryptography (and such AES) in their system which is, in the worst case, even required by multiple components within a complex System-on-a-Chip (SoC) architecture. Since many of those components of the SoC are provided by different vendors with no mutually trusted relationships, it is not uncommon that every security-critical SoC component has a separate security subsystem with an individual AES core. In particular, for advanced security functions such as feature activation and IP licensing of components some schemes may even require per-component availability of an individual AES [11].

If the AES core is only considered a rarely used and supplementary function, it is essential to keep it resource consumption as low as possible. But though being a low-cost supplementary function, aspects of physical security of an AES implementation in hardware need to be considered. Any cryptographic hardware implementation that is potentially physically exposed to an attacker must include countermeasures against side-channel attacks (SCA) [18]. Most countermeasures against SCA require randomness to mask or hide the exploitable side-channel information, hence this demands for the availability of a connected random number generator. Block ciphers can be used as Cryptographic Pseudo Random Number Generators (CPRNG) in order to expand random seeds obtained from an entropy source. In this sense, all supplementary cryptographic implementations should be lightweight, physically protected and as far as possible self-contained and include a facility to generate the required randomness e.g., for key generation or to feed physical countermeasures.

Contribution: In this work, we present a threefold contribution. First we propose an extremely lightweight AES-128 implementation optimized for recent Xilinx FPGA generations. To the best of our knowledge, our proposal is the smallest AES-128 design reported so far and fits into 21 slices of a Spartan-6 with no additional block memories. Our approach takes solely advantage of the *Distributed Memory* instances of recent Xilinx FPGAs that can be perfectly matched with an 8-bit serialized AES architecture. Second, we show that our basic concept can be easily enhanced to provide an AES-based CPRNG that can be, finally, used to build a side-channel protected, self-contained AES-128 encryption architecture occupying not more than 28 slices. Despite the minimal resource requirements, a single encryption with our self-contained, protected AES-128 core takes 1471 cycles that translates to the throughput of 4.35 Mbit/s. We also want to emphasize that all of our proposed designs include the key schedule and hence are fully functional in order to perform AES encryptions even with frequent rekeying. Eventually, we also performed practical side-channel evaluations of our self-contained AES-128 encryption architecture and show its significantly increased resistance against physical attacks while

still maintaining its lightweight and low-area properties.

Note that our compact encryption architecture easily outperforms any PicoBlaze [25] instance (an 8-bit soft-core microcontroller tailored for Xilinx FPGAs) that runs a software AES. Such an instance requires 26 slices and additional *Block Memory* (BRAM) in its minimal configuration and more than 13546 cycles [12] per encryption.

Outline: The remainder of this article is organized as follows: Sec. II provides and summarizes relevant previous work while Sec. III outlines basic Xilinx FPGA features, the AES-128 encryption algorithm, and Side-Channel Analysis in general. Sec. IV describes initial design considerations and decisions, before our basic encryption architecture and several enhancements are presented in Sec. V and evaluated and compared in Sec. VI. Results of our practical side-channel evaluations are given in Sec. VII. Finally, our work concludes in Sec. VIII.

II. Previous work

We briefly summarize previous results of relevance to this work and our contribution. Since there is a wealth of publications addressing AES architectures on reconfigurable hardware, e.g., [7], [15], [21], [20], [14], [12], [6], [10], [5], we restrict the discussion of previous works to the most relevant ones. As one of the first AES implementation optimized for resource-limited applications on reconfigurable hardware, in 2005, Good and Benaissa [12] proposed an Application Specific Instruction Processor (ASIP) based on Xilinx' PicoBlaze [25] soft-core microcontroller. This publication was followed by many more, but one of the first architectures that was designed for the newer FPGA generations starting with Virtex-5 is given in [5], where for the first time a minimal S-box implementation of 8 slices was introduced. In 2012, Chu et. al. [8] picked up the idea of Chodowiec and Gaj [7] to use Shift Register Look-up Tables (SRL16) in order to store and process intermediate values during the round computation. For their final design, the authors can report a size of 80 slices for an encryption architecture on a Spartan-6 FPGA, which is the smallest memory-free AES-128 FPGA implementation to date.

III. Preliminaries

In this section we will briefly recap the basic elements inside Xilinx FPGAs that are relevant for the optimizations done in this work. Then, we introduce the Advanced Encryption Standard (AES) and its encryption algorithm before briefly outlining the general concept of Side-Channel Analysis (SCA) and common countermeasures against side-channel attacks.

A. FPGA Elements

Starting with the Virtex-5 generation of Xilinx FPGAs, the Configurable Logic Blocks (CLB) are nowadays populated with slices that provide different features for various use-cases:

Slice-X: The basic slice architecture provides four independent Look-Up Tables (LUT) in order to implement arbitrary 6-to-1 or 5-to-2 Boolean functions. Since all output signals can be synchronized to a clock signal, each slice provides 8 flip-flop instances.

Slice-L: Enhanced slice versions with special abilities for arithmetic operations and logic handling are called Slice-L. Using additional internal multiplexers, LUTs can be combined to implement 7-to-1 or 8-to-1 Boolean functions.

Slice-M: The most powerful slices augment the capabilities of the Slice-L version by memory features called *Distributed Memory* turning the configuration memory of LUTs into user-accessible storage (either 256-bit memory or 128-bit shift registers) with several different user-defined configurations.

B. Advanced Encryption Standard (AES)

Although we assume that the AES is a well-known block cipher, we now briefly review its details for the sake of self-containedness of this work. AES is a symmetric key-alternating block cipher with 128-bit block size standardized for three different key sizes (128-bit, 192-bit and 256-bit) based on a Substitution-Permutation Network (SPN) with 10-14 rounds. Each round can be subdivided into atomic operations that work byte-oriented using field arithmetic over the Galois Field $GF(2^8)$. The following section shortly introduces the AES-128 variant of the encryption algorithm and presents necessary background information.

1) Round Function of the AES Algorithm: The AES round function is designed as Substitution-Permutation-Network (SPN) operating on an 128-bit internal state arranged as a 4×4 matrix of bytes. During the encryption process, the state is transformed using four basic operations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*.

SubBytes: The byte substitution S is derived using inversion in $GF(2^8)$ and a subsequent affine transformation, such that $S(x) = M \cdot x^{-1} + c$. The substitution can be computed on-the-fly or it can be precomputed and stored as look-up table (called S-box) to reduce computational overhead. It is applied to the state by substituting all bytes independently.

ShiftRows: This operation performs a cyclic byte-wise left rotation of each row R_i of the state matrix. In general, the i-th row is shifted i positions to the left.

MixColumns: The *MixColumns* operation considers each column of the state matrix as a vector of polynomials in $GF(2^8)$ and performs a vector-matrix multiplication for all columns independently using a standardized matrix with small constants.

AddRoundKey: Finally, the round key generated by the key expansion process is added to the state matrix in a byte-wise fashion using XOR-operations.

For a full AES-128 encryption, ten round functions are applied repeatedly in order to transform a plaintext into a ciphertext. However, the first round function is preceded by

an additional *AddRoundKey* operation (considered as *key-whitening*) using the initial key. The ten round functions then are identical except for the last round which omits the *MixColumns* operation.

2) Key Schedule of the AES-128 Algorithm: The update function of the 4×4 key matrix is applied column-wise, starting with the leftmost column by rotating it one byte position and applying four S-boxes. Additionally, a round constant RC_i (computed as polynomial x^i in $GF(2^8)$) is added to the first byte. In the following, the remaining columns are derived by adding the recently computed column with the next column.

C. Side-Channel Analysis (SCA)

Side-Channel Analysis (SCA) exploits unintended information leakage of a cryptographic device related to an internal state (such as secret intermediate values), e.g., by inspecting timing behavior [16], power consumption [17] or electromagnetic emanations (EM) [1].

1) Power Analysis: In particular, the inspection of leakage due to the power consumption of a cryptographic device has been studied intensively by the academic community. Broadly speaking, power analysis can be classified as either Simple Power Analysis (SPA) or Differential Power Analysis (DPA).

Simple Power Analysis: This kind of attack aims at identifying the leakage in order to extract the key or a secret intermediate value by analyzing (a small number of) power traces usually searching for key-dependent patterns. Hence, significant experience or special knowledge of the cryptographic device and its internal architecture are required.

Differential Power Analysis: In contrast, DPA exploits data dependencies in the power consumption by processing measured traces using statistical methods to identify the correct key among a set of key hypotheses. A prevailing method, called Correlation Power Analysis (CPA), uses correlation coefficients to identify the correct key. However, a significant number of power traces are required for a successful attack.

2) Countermeasures: In general, SCA countermeasures aim to avoid the leakage of information using various approaches. Established countermeasures to counteract power analysis are commonly classified as *hiding* or *masking*.

Hiding: Hiding countermeasures aim at burying the exploitable signal within (random) noise. Hence, the SNR is decreased by either increasing the noise or reducing the signal. Common approaches for hiding are shuffling of instruction execution order, insertion of dummy instructions or even dedicated logic styles e.g., [22], [23] (in particular for customized hardware implementations).

Masking: Instead of decreasing the SNR, masking countermeasures randomize the power consumption of a device in order to break the data-dependencies. Mostly, these countermeasures are applied on an algorithmic level by masking the input and output values of crucial functions with random values, hence randomizing the power consumption depending on the processed data.

IV. Design Considerations

Usually, lightweight cryptographic architectures are constrained by the storage of intermediate states and the width of their internal data path. Fortunately, AES mainly relies on byte-wise operations and inherently supports an area efficient 8-bit data path. Besides, modern FPGAs are typically rich in memory elements, in particular considering *Distributed Memory* primitives. Observing that modern Xilinx FPGAs offer up to 256-bit *Distributed Memory* per slice yields the opportunity to push the limits for resource-constrained implementations on FPGAs further by reducing the area of state registers. Moreover, we could identify the S-box and the *MixColumns* operation as most area and logic consuming operations that require some special effort in order to implement low-area instances. For the S-box, the smallest implementation found in the literature (based on 6-input LUTs), still requires at minimum 8 slices which leaves no room for optimization. However, the *MixColumns* mainly applies byte-level Galois Field arithmetic (although operating on 32-bit columns) and can be merged easily with the key addition operation. By this, the total round function logic and its critical path finally could be reduced to the S-box and a subsequent *MixColumns* unit. Since each round requires different sub-keys that have to be derived on the fly, a complete AES-128 encryption core requires additional logic for key expansion. Fortunately, the key update unit can be merged entirely with the round function logic since only a few byte-wise substitutions using the S-box as well as Galois Field arithmetic (provided by the *MixColumns* unit) are required to derive a new sub-key. Thus, by sharing resources we can implement an ultra-lightweight AES-128 encryption core including key expansion in a handful of slices.

A. Protection against Physical Attacks

Although designed to be lightweight, our design should include countermeasures to thwart physical side-channel attacks. We therefore employ a design methodology that easily allows shuffling of the byte execution order as hiding-based countermeasure. In particular, the individual access to stored bytes (thanks to the utilization of *Distributed Memory*) provides the option to extend our design quite naturally with this countermeasure.

B. Pseudo Random Number Generation

Commonly, True Random Number Generators (TRNG) are unable to provide sufficient randomness required by many side-channel countermeasures in very short time. Hence, in practice a hybrid approach is used, where a TRNG can provide a random seed which then is expanded using a PRNG. Often, block cipher primitives are used to extract the pseudo-randomness from an entropy source since their results provide all necessary statistical properties, then usually considered as

Cryptographic Pseudo Random Number Generator (CPRNG). Our self-contained encryption engine offers straightforward protection against side-channel attacks and is independent of external RNGs (after an initial random seed has been provided) since encryption runs are interleaved with the generation process of pseudo randomness using the AES core.

C. Modes of Operation and Decryption

Our architecture is designed to support encryption but no decryption. However, encryption is always used in a specific mode of operation, i.e. the encryption core is embedded into an particular configuration that satisfies application-specific requirements, such as blockwise, parallel encryption. An extremely popular mode of operation is the Counter Mode (CTR), which turns the original block cipher into a stream cipher. The advantage of this mode is that both encryption and decryption can be achieved by using an encryption-only core that generates blocks of a key stream that are added to the plaintext and the ciphertext, respectively. The key stream is generated using a random nonce concatenated with a counter that is incremented for every block. Taking this into account we assume that an encryption-only core should be sufficient for most applications with need for lightweight cryptography.

V. Implementation

In this section, we present details of our basic encryption architecture. Although this implementation is designed for the Xilinx Spartan-6 FPGA family, it can be easily ported to any other FPGA family providing 6-input LUTs and a similar integrated memory option. By aggressive sharing of components we limit the resource requirements of the basic implementation to 15 slices for all functional blocks and additional 6 slices are necessary to control the entire encryption process. In a second step, we extend our basic core design with a fundamental protection feature against side-channel analysis by introducing random shuffling of the byte execution order during round computations. We like to highlight that due to the architecture of the basic core this is possible by only slightly modifying the architecture. The required randomness to derive the permutation of the execution order can be provided either by an (additional) external RNG or even by the AES-128 encryption core itself that is employed as CPRNG after being provided with an initial (random) seed. We remark that even with the CPRNG option, the introduced overhead is still minimal and the resource requirement of the basic core is only slightly increased.

A. Basic Encryption Core

The overall architecture of our AES-128 encryption and key update function is shown in Figure 1. All state and round key information is stored within two *Distributed Memory* components.

1) Initialization: Prior to each encryption, registers (Figure 1: STATE and KEY) are initialized with the plaintext and key, respectively. Both registers are implemented as 256-bit *Distributed Memory* (configured as 32×8-bit memory primitives). Plaintext and key bytes have to be provided alternately, for two clock cycles each and while the bytes of the key are stored unaltered, the round state register is initialized with the result of the *key-whitening* operation. In total, the initialization phase for a new encryption requires 65 clock cycles.

2) Key Schedule: Before a round function can be executed, the key update function has to be applied to the key register state, in order to compute the sub-key. This process is performed in a byte-sequential flow of 50 clock cycles merged with the round function logic and requires 4 S-box computations and 17 Galois field additions (one constant addition and 16 byte additions). Unfortunately, due to the computation using byte-wise operations, data dependencies are introduced which restrict the degree of flexibility of this process (with respect of shuffling the execution order). However, since the round key occupies only half of the register, the second half is used to store all round constants.

3) Round Function: In the following, we describe the integration and realization of the atomic AES operations within our developed encryption architecture. Benefiting from previously described design considerations, we could construct a hardware architecture using a minimal number of resources and slices of a selected Spartan-6 device which requires 97 clock cycles per round.

ShiftRows: This operation can be realized by using address translation for either the loading or the storing addresses of the state register. For our design, we decided to change the storing address according to the *ShiftRows* function. Note, that this is also the reason for inherent shuffling support, since it allows to easily randomize the execution order of the bytes only by adapting the address translation function using a random permutation.

SubBytes: Trying to optimize and minimize the S-box as far as possible, we could identify the look-up table approach to result into the most compact S-box representation for FPGAs using 6-input LUT technology. Still, the S-box requires 8 slices, whereby each slice realizes a single 8-to-1 Boolean function using internal multiplexers (requires Slice-L or Slice-M).

MixColumns: When breaking down the *MixColumns* operation to its basic functions, it can be observed that only few different operations have to be implemented in order to support the entire operation: multiplications with small constants (i.e., `0x01`, `0x02` and `0x03`) as well as simple XOR addition. However, the implementation of the *MixColumns* operation is challenging, since the operation is executed column-wise rather than byte-wise. In order to serialize the execution of *MixColumns*, we have to introduce an 8-bit wide register to hold intermediate values. This register along with preceding logic implements some accumulator-like logic that provides four simple operations:

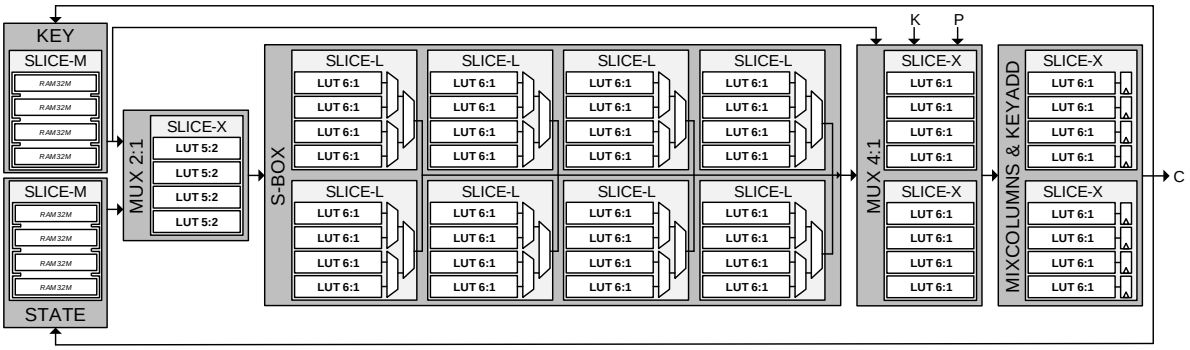

Fig. 1: Architecture overview of the basic AES-128 encryption core: The round function and key schedule are implemented in 15 slices, the remaining 6 slices are necessary for control logic (omitted in the figure for the sake of clarity).

$r_i = x_i$ (*write x into the register r*)
$r_i = r_{i-1} + 01 \cdot x_i$ (*add x multiplied by 01 to r*)
$r_i = r_{i-1} + 02 \cdot x_i$ (*add x multiplied by 02 to r*)
$r_i = r_{i-1} + 03 \cdot x_i$ (*add x multiplied by 03 to r*)

Besides a 2-bit control signal to select the current operation, a small 2-bit counter is required to decide which column is processed currently in order to select and load the required bytes in the correct order.

AddRoundKey: Before the *MixColumns* operation can be started, the register of the MixColumns unit is initialized with the current sub-key byte in order to merge the key addition with the *MixColumns* operation. The results from the Galois Field multiplications are added to the register so that it holds the accumulated result of both operations.

4) Last Round: After applying the *SubBytes* operation for the final round, the round key and the four additions and multiplications of the *MixColumns* operation are skipped. Due to the inherent latency of the additional intermediate register within the MC unit, the correct ciphertext bytes are returned every second clock cycle, hence the last round is computed in 33 clock cycles.

5) Control Unit: Decreasing the complexity of round logic usually comes at cost of increasing the complexity of control logic. In general, the control logic is in charge of controlling the addressing of the internal registers, the input selection of the multiplexers, selection of the current *MixColumns* sub-operation and the state transitions of the internal finite state machine (FSM) in order to realize the entire AES encryption. Therefore, it implements several counters (round and current byte counter) which can be implemented in a single slice. Besides the internal FSM which has 32 different states and requires five flip-flops to store and five LUTs to update the state. Most complex part of the control logic is the generation of the control signals including the addressing of the state registers. In particular the address generation is costly since it considers the *ShiftRows* operation. The remaining control signals are generated using a microcode-like approach, where the control signals are stored in Read-Only Memories (ROM) and are selected depending on the current state of the FSM.

B. Adding Side-Channel Protection

A common and basic way to hide leakage within measured power consumption is shuffling of operations. A random execution order makes it more difficult for an attacker to predict the internally processed data of the device and thus hampers side-channel analysis of the implementation. Fortunately, shuffling of the byte update process during the round computation can be implemented by randomizing internal addresses (as discussed before) and is supported inherently by our design. In the following section we first explain how to determine random permutations with modest area overhead and in a second step, we present the extension of our basic encryption architecture which implements the proposed shuffling countermeasure.

1) Permutation Generation: A uniform permutation $Perm$ of any n-element sequence Seq can be generated using a linear-time algorithm that applies n swapping operations to Seq. The algorithm starts with $i = 0$ and iterates over all elements $s_i \in Seq$ with $i \leq n-2$ and swaps the selected element with a random element of the remaining part of the sequence i.e., $\{s_i, ..., s_{n-1}\}$. However, sampling random numbers from $\{i, ..., n-1\}$ is not trivial and not feasible for lightweight implementations, because it requires either modulo operations or algorithms with probabilistic runtime. Hence, we decided to follow the approach of Veyrat-Charvillon et al. [24] who proposed to sample from $\{0, ..., n-1\}$. According to their investigation, this approach results in a slightly biased permutation, but they conclude that in the context of side-channel analysis this can be neglected. Following this approach, we store two initial permutations $Perm_0 = \{0, ..., 15\}, Seq = \{0, ..., 15\}$ in a 64×4-bit memory. In particular, Seq is necessary to realize operations that cannot be shuffled (such as the key update function). The memory is succeeded by a 4-bit register which holds intermediate values during the swapping operation of the permutation algorithm that is interleaved completely with the phase of receiving and storing the initial plaintext and key bytes.

2) Shuffling of Round Update Order: In general, a new permutation $Perm_i$ is derived from the previous permutation $Perm_{i-1}$ stored in the memory using 64 bit of randomness. The permutation process is executed in parallel to the *key-*

Design/ Variant	Device	Implementation						Performance			
		Mode	*Datapath*	*Logic*		*Memory*		*Clock*	*Period*	*Throughput*	*Throughput/Area*
		(Enc/Dec)	*(Bit)*	*(LUTs)*	*(Slices)*	*(FFs)*	*(BRAM)*	*(MHz)*	*(Cycles)*	*(Mbps)*	*(Mbps/Slice)*
Bulens et al.[5]	Virtex-5	Enc	128	-	400	-	0	350	11	4,072	10.18
Good and Benaissa [12]	XC2S15	Enc/Dec	8	-	124	-	2	67	3691	2.2	0.02
PicoBlaze [12]	XC2S30	Enc/Dec	8	-	119	-	2	90	13546	0.71	0.01
Chodowiec and Gaj [7]	XC2S30	Enc/Dec	32	-	222	-	3	55	44	166	0.75
Chu and Benaissa [8]	XC3S50	Enc	8	-	184	-	0	46	160	36.51	0.20
Rouvroy et al. [20]	XC3S50	Enc/Dec	32	-	163	-	3	72	46	208	1.28
Chu and Benaissa [8]	XC6SLX4	Enc	8	-	80	-	0	73	160	58.13	0.73
This work / basic	XC6SLX4	Enc	8	84	21	23	0	105	1471	9.12	0.43
This work / shuffling	XC6SLX4	Enc	8	94	24	29	0	90	1471	7.82	0.33
This work / PRNG	XC6SLX4	Enc	8	112	28	36	0	75	1471	4.35	0.16

TABLE I: Comparison of selected implementations of AES for FPGAs

whitening of each encryption. During the encryption, the address signals are reconnected to the permutation memory instead of to the state registers. Then, the permutation generator serves as the address translation unit which changes the address signals according to the current permutation $Perm_i$ and thus shuffles the byte update process after the *MixColumns* operation of each round. Otherwise, the second sequence Seq is applied in order to avoid the shuffling, e.g., in case the key schedule is executed which requires a fix execution order due to data dependencies.

C. Pseudo Random Number Generation

Since symmetric block ciphers provide good statistical properties and are easy and fast to evaluate, they allow to generate larger amounts of randomness starting from a relatively small seed. In order to design an AES implementation as lightweight as possible which still provides basic protection against side-channel attacks, sufficient amounts of randomness are required. Hence, an obvious approach is to reuse our AES core itself in order to extract the required randomness from an external random seed and build a self-contained side-channel protected lightweight AES instance. Since 64-bit fresh randomness are required for every encryption while the AES core can provide 128-bit at once, a small 2-bit counter controls switching to RNG mode after two consecutive encryptions. During the operation as RNG, the AES core returns a busy signal and does not accept encryptions.

VI. Results

Our basic encryption architecture is implemented in only 21 slices[1] using 100% of the LUTs and 14% of available flip-flops within the slices. The design can operate at a maximum clock frequency of 105 MHz and requires 1471 clock cycles for a full encryption, i.e., the maximum throughput is 9.12 Mbit/s. Side-channel protection increases (due to the permutation generation unit) resource consumption by additional 10 LUTs and 6 flip-flops to a final size of 24 slices. Unfortunately, the critical path is affected, dropping the maximum clock frequency to 90 MHz and the throughput to 7.82 Mbit/s. Finally, our self-contained AES encryption core again increases the resources consumption by 18 LUTs and 7 flip-flops to a final size of 28 slices. Note, however, that this design still provides the shuffling countermeasure. The final clock frequency is 75 MHz but due to the interleaving of encryption and RNG mode, the throughput for encryptions is further reduced by 33% to 4.35 Mbit/s.

[1] All results were obtained after place-and-route based on a Xilinx Spartan-6 XC6SLX4 using Xilinx ISE 14.7. The dense packing into a minimum number of slices was achieved by applying several constrains and placing almost every component by hand.

A. Comparison

In Table I we list previous results of AES implementations for FPGAs reported in the literature. Note, however, that many of the listed implementations were designed for older Spartan-3 devices that do not provide comparable features, such as 6-input LUTs and 256-bit *Distributed Memory*and thus do not allow a fair comparison. Still, some of the related works have been implemented for both, Spartan-3 and Spartan-6 device families. In particular, recent work in [8] provides details for a Spartan-6 based design. The authors optimized the application of Shift Register instances and report final area numbers of 80 slices on an XC6SLX4 device. In addition, although implementing an 8-bit wide data path, the data path for the MixColumns operation is extended to 32-bit at cost of higher resource consumption. Still, the reported throughput of the design is much higher than our moderate performance due to the smaller number of clock cycles required for a single encryption.

VII. Side-Channel Analysis

For our practical side-channel evaluations, we used another Spartan-6 XC6SLX75 embedded on a SAKURA-G board [13]. The traces of the instantaneous power consumption have been measured by means of a LeCroy digital oscilloscope which monitored the voltage drop over a 1Ω resistor placed in the V_{dd} path. All traces have been recorded via the embedded amplifier of the SAKURA-G platform at a sampling rate of 500 MS/s. In order to simulate best-case scenarios for an attacker, the design is running at a low frequency of 3 MHz in order to reduce noise caused by overlapping power traces. We opted to perform a key-recovery attack on the S-box input of

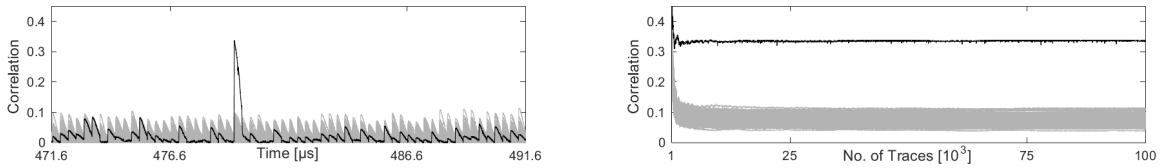

Fig. 2: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round *without* shuffling of the byte execution order).

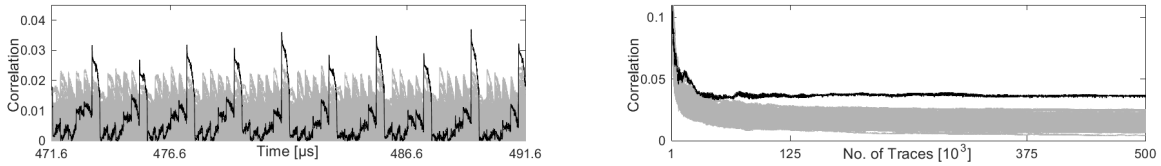

Fig. 3: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round *with* shuffling of the byte execution order).

the last round. However, since our design is self-contained in terms of random number generation, we only provided random plaintexts and observed corresponding ciphertexts while our design was protected by means of shuffling.

A. Evaluation under Lab Conditions

In a first step, our design is evaluated in an isolated environment assuming an optimal environment for any side-channel observer. In particular, we performed two different measurements: first, we evaluated the side-channel resistance of our unprotected design (without shuffling the byte execution order) and second, we examined the resistance and gain in security when activating our integrated side-channel countermeasure.

1) Unprotected: For verification of the setup and implementation, an initial measurement (without applying shuffling to the byte execution order) is conducted. In order to recover the secret key, we performed a CPA and used the Hamming weight power model as distinguisher. The results for attacking the S-box input of the last round guessing the fifth key-byte are exemplary shown in Figure 2. For this attack we measured up to $100\,000$ power traces, but the correct key byte could already be revealed after about 200 traces.

2) With Shuffling Countermeasure: After the checking the setup and implementation for correctness, our shuffling countermeasure is applied during operation for a second measurement. Figure 3 provides the result of the CPA (using the same setting as for the preceding evaluation). This time, we measured $500\,000$ power traces for random inputs but fixed key. It is obvious that shuffling clearly improves the resistance of the design against side-channel attacks, in particular the S-box computation is randomly performed at different time instances. However, perfect resistance still cannot be achieved but a successful key-recovery can be impeded when additional burden (and countermeasures for the attacker) are present. Without this, a key-recovery is possible requiring about $50\,000$ power traces before the correct key hypotheses could be clearly distinguished.

B. Evaluation for Embedded Applications

In order to simulate a more realistic environment in which our AES-128 encryption core is embedded, we implemented an additional core that is performing arithmetic computations in parallel. This core occupied around $9\,000$ slices (80% of our target device). Again, we performed two different measurements in order to evaluate the impact of the side-channel countermeasure.

1) Unprotected: Obviously, when increasing the arithmetic noise of the measurements, the Signal-to-Noise Ratio (SNR) decreases which can be seen in the reduced correlation coefficients. A successful key-recovery attack is possible, as shown in Figure 4, but now requires around $5\,000$ power traces.

2) With Shuffling Countermeasure: Eventually, we again implemented the shuffling of the byte execution order while this time, in contrast to the side-channel evaluation under lab conditions, we simulated arithmetic noise using a second core operated in parallel. The evaluation results of this measurement can be seen in Figure 5. For a successful and unambiguous key-recovery, at least about $800\,000$ power traces have to be recorded.

Our results show that the spending of 7 additional slices for countermeasures can improve the resistance to physical attacks such as DPA or CPA significantly, in particular under practical conditions considering background noise introduced by additional cores running in parallel. Lightweight characteristics and properties of our design are still preserved although the encryption core is reused for random number generation. But even if side-channel protection is not mandatory, our architecture can still be used to generate pseudo randomness which then can be used for other purposes.

VIII. Conclusion

In this paper we proposed three extremely lightweight architectures for AES-128 encryption for recent Xilinx FPGAs. Our

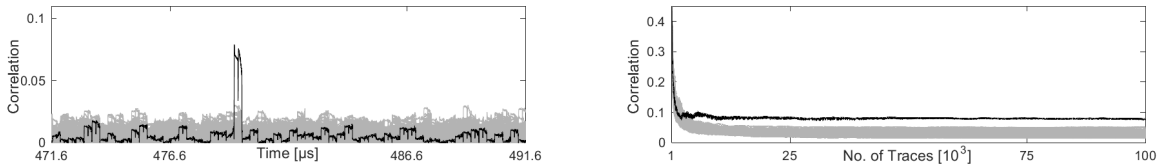

Fig. 4: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round *without* of shuffling the byte execution order) with arithmetic noise.

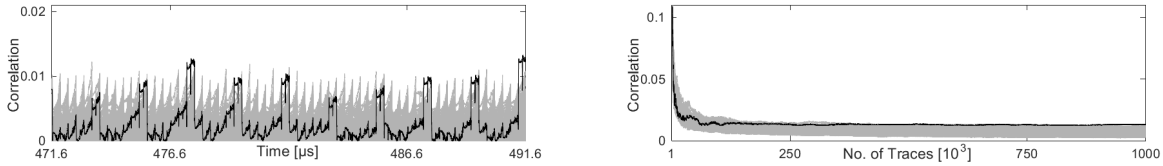

Fig. 5: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round *with* shuffling of the byte execution order) with arithmetic noise.

designs extensively exploit the 256-bit *Distributed Memory* provided within a single slice of modern Xilinx FPGAs as well as the availability of 6-input LUTs. The presented architectures provide throughputs between 9.12 and 4.35 Mbit/s which are sufficient for many applications with moderate requirements on performance but strict constraints on the resource consumption for cryptographic instances. Furthermore, we showed that side-channel protection and randomness generation can be included in the same core while maintaining lightweight properties of the implementations required by many security applications.

REFERENCES

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002, Revised Papers*, 2002.

[2] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[3] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, Proceedings*, 2007.

[4] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In *Advances in Cryptology - ASIACRYPT 2012, Proceedings*, 2012.

[5] P. Bulens, F. Standaert, J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In *Progress in Cryptology - AFRICACRYPT 2008, Proceedings*, 2008.

[6] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable memory based AES co-processor. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings*, 2006.

[7] P. Chodowiec and K. Gaj. Very Compact FPGA Implementation of the AES Algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2003, Proceedings*, 2003.

[8] J. Chu and M. Benaissa. Low area memory-free FPGA implementation of the AES algorithm. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[9] J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Nessie proposal: Noekeon, 2000.

[10] S. Drimer, T. Güneysu, and C. Paar. DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs. In *16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008*, 2008.

[11] S. Drimer and M. G. Kuhn. A Protocol for Secure Remote Updates of FPGA Configurations. In *Reconfigurable Computing: Architectures, Tools and Applications, 5th International Workshop, ARC 2009, Proceedings*, 2009.

[12] T. Good and M. Benaissa. AES on FPGA from the Fastest to the Smallest. In *Cryptographic Hardware and Embedded Systems - CHES 2005, Proceedings*, 2005.

[13] H. Guntur, J. Ishii, and A. Satoh. Side-channel AttacK User Reference Architecture SAKURA-G. In *GCCE 2014*. IEEE Computer Society, 2014. http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[14] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. In *FCCM 2004, Proceedings*, 2004.

[15] K. U. Järvinen, M. Tommiska, and J. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *FPGA*, 2003.

[16] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO 1996, Proceedings*, 1996.

[17] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*. Springer, 1999.

[18] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[19] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Advances in Cryptology - EUROCRYPT 2011, Proceedings*, 2011.

[20] G. Rouvroy, F. Standaert, J. Quisquater, and J. Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications. In *ITCC 2004, Volume 2*, 2004.

[21] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In *Cryptographic Hardware and Embedded Systems - CHES 2003, Proceedings*, 2003.

[22] K. Tiri and I. Verbauwhede. A Dynamic and Differential CMOS Logic Style to Resist Power and Timing Attacks on Security IC's. *IACR Cryptology ePrint Archive*, 2004.

[23] K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *DATE 2004, Proceedings*, 2004.

[24] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F. Standaert. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In *Advances in Cryptology - ASIACRYPT 2012, Proceedings*, 2012.

[25] Xilinx. PicoBlaze 8-bit Embedded Microcontroller User Guide, 2011.