Journal of Signal Processing Systems manuscript No.
(will be inserted by the editor)

Cryptographic Algorithms on the GA144 Asynchronous Multi-Core Processor

Implementation and Side-Channel Analysis

Tobias Schneider · Ingo von Maurich · Tim Güneysu · David Oswald

Received: date / Accepted: date

Abstract Pervasive computing has turned many ordinary commodity products to smart and digital computing devices. Though these devices are mostly equipped with low-cost processors offering limited computing power, they are often requested to handle user-sensitive data. This evidently calls for the integration of different security services that typically involves computationally expensive cryptography. In this context, lightweight cryptographic constructions came recently up to minimize the computational burden on such constrained devices. Unfortunately, many of those constructions were too simplistic to preserve long-lasting confidence in their security. Therefore we aim for another approach in this work and implement standardized and well-established cryptography on an alternative, lightweight platform, namely an asynchronous GA144 ultra-low-powered multi-core processor with 144 tiny cores. We demonstrate that symmetric and asymmetric cryptography such as AES and RSA can be realized on this low-end device. With energy consumption being as low as 0.63 μJ and 22.3 mJ, this platform achieves a performance of 38 μs and 462.9 ms per AES and RSA operation, respectively. This translates to an energy consumption and computation time that is significantly lower than many lightweight implementations reported so far. We finally emphasize that this low-power and asynchronous operation of cryptography does not eliminate the threat of physical attacks, in particular power attacks. We evaluate the side-channel resistance of our design and identified that less than 5,000 measurements are already sufficient to fully recover the 128-bit key of the unprotected AES implementation.

Keywords GA144 · asynchronous processor · low-power · AES · RSA · implementation · multi-core · side-channel analysis

Tobias Schneider · Ingo von Maurich · Tim Güneysu · David Oswald
Universitätsstr. 150
44780 Bochum
Germany
Tel.: +49-234-3224626
E-mail: {tobias.schneider-a7a,ingo.vonmaurich,tim.gueneysu,david.oswald}@rub.de

1 Introduction

Pervasive computing has raised the bar to integrate complex systems on (nearly) invisible and small devices, which often run on battery or even completely lack a power supply and harvest energy from an electro-magnetic (EM) field. In addition, these devices increasingly handle private data that has to be protected using computationally expensive security services to ensure confidentiality and integrity.

Many works so far adapted lightweight cryptographic algorithms to small systems, often sacrificing security and/or speed to achieve low-power cryptographic services. Standardized cryptographic algorithms are often just too bulky to fit on such small devices, especially public-key operations with millions of involved multiplications are problematic.

We highlight a different approach in this work and implement standardized and established cryptographic algorithms on Greenarrays' GA144 lightweight asynchronous multi-core processor. Our main goal is to investigate if there are more efficient processor platforms for cryptographic operations than those that are currently developed and distributed by the market leaders AMD, ARM, AVR, and Intel.

An important aspect of the security of embedded systems is the susceptibility towards implementation attacks [20] that—in contrast to purely mathematical cryptanalysis—exploit weaknesses of the physical realization of a cryptographic operation. While (active) *fault injection* is based on disturbing the computation on a device to extract secrets, (passive) *side-channel analysis* utilizes physical leakages, e.g., the power consumption or the EM emanation, to deduce cryptographic keys. Thus, the question arises to which extent an asynchronous processor like the GA144 can be attacked using side-channel analysis. In this paper, we address this issue, exemplarily evaluating one of our implementations of the AES.

1.1 Contribution

In this paper we investigate how established cryptographic algorithms perform on an asynchronous multi-core processor with 144 simplistic cores and present the first implementations of the standardized symmetric AES block cipher and the asymmetric RSA cryptosystem with a focus on maximizing energy-efficiency.

Even though the capabilities of the simplistic cores of the GA144 are very limited (e.g., only 64 words of program memory per core), we show that by combining multiple cores even complex cryptographic operations can be performed at moderate speeds while keeping the energy consumption low at the same time.

Our implementations achieve a decent performance of 38 μs and 462.9 ms per AES and RSA operation and the consumed energy is as low as 0.63 μJ and 22.3 mJ, respectively. Hence symmetric and asymmetric cryptography is not just feasible on such a low-end and unclocked device but the two important metrics computation time and energy consumption are significantly lower than many lightweight implementations reported so far.

Finally, we also investigate the hardware security of the GA144, analyzing the susceptibility towards side-channel attacks. To this end, we focus on one implementation of the AES and show that the full 128-bit secret key can be extracted with less than 5,000 measurements of the EM emanation of the device.

1.2 Related Work

Hardly any other implementations are available for the GA144 as it is a very atypical and new processor. An implementation of the outdated hash function MD5 was published by Greenarrays and a pipelined implementation of the hash function SHA-256 has been announced. Due to the lack of implementations of AES and RSA for this platform, we compare our results to other processor platforms and ASIC implementations running AES and RSA.

With respect to implementation attacks and side-channel analysis in particular, following the initial discovery by Kocher et al. [15], numerous real-world devices have been shown to be vulnerable to this threat. Examples include microcontroller by virtually all major vendors [20], Mifare DESFire MF3ICD40 contactless smartcards [26], security mechanisms of various FPGAs [22–24, 30], and electronic locking systems [27]. The use of asynchronous (and balanced) circuits to protect smartcards against side-channel attacks is described in [6]. To our knowledge, the side-channel susceptibility of the asynchronous architecture of the GA144 has not been previously examined.

1.3 Outline

This work is organized as follows: we first describe the novel architecture of the GA144 and explain its programming paradigm in Section 2. Next, we present both our implementations of AES and RSA in Sections 3 and 4, respectively. We present our implementation results in Section 5. In Section 6, we analyze the AES implementation on the GA144 from a side-channel point of view. Finally, we conclude our work in 7.

2 Processor Architecture of the GA144

In the following we give a brief description of the internal structure of GreenArrays GA144 multi-core processor. We start by introducing the F18A computing node as the foundation of the processor, explain how 144 of these cores are interconnected to form the GA144 and give a short intuition on how to program these chips.

2.1 The F18A Computing Node

When designing the F18A, GreenArrays aimed to create a small and inexpensive computing node which is both fast and energy-efficient. The processor operates on a word size of 18 bit and is equipped with 64 words of RAM and ROM. Furthermore, every F18A possesses stacks, registers, and (optionally) various kinds of external I/O [8].

The processor operates completely asynchronously from other connected devices. While basic instructions are executed in about 1.5 ns without instruction fetches, memory operations can take up to 5 ns. Since one of the main optimization goals of the processor is energy-efficiency, these executions only cost 7 pJ and

the F18A can be suspended even in mid-instruction. When suspended, a small leakage of about 100 nW keeps the power consumption at a minimum [10].

The two available stacks are a ten-element data stack and a nine-element return stack. The top two elements of the data stack are represented by **T** and **S** and the remaining eight form a circular memory. The return stack is build similarly, the top of the stack is stored in register **R** and the following eight elements are stored in so-called circularly addressed registers. Additionally, every F18A has an instruction register **I**, a program counter **P**, a general-purpose read/write register **A**, and a write-only register **B** [8].

There are four different ways for a F18A to communicate with external devices. These are not available for every computing node and are realized depending on its position on the GA144 chip. Speaking in major classes of I/O, a F18A can utilize general purpose input/output (GPIO), analog I/O, 18-bit parallel buses, and a high speed serializer/deserializer (SerDes) [11].

The data flow between F18A nodes is realized using communication ports which enable the nodes to exchange data at high speed. Additionally to exchanging data, instructions can be transferred over these ports to accomplish port execution. This means the receiving F18A executes them directly and, since the program counter is not incremented, a whole instruction stream can be processed in short time. Moreover, this technique can be utilized to save program memory in the receiving node because no additional code is required for port execution. In the most extreme case a node has no program code at all and just waits for instructions to be fed to it from the outside. As a result the program memory of this node is empty and can be used to store other data. Another application for this is reprogramming a F18A during runtime by rewriting the program memory using port execution and then jumping to an address to start the program.

2.2 The GA144 Multi-core Processor

The GA144 consists of 144 F18A nodes which are arranged in an 18x8 array. The F18As are connected with their adjacent nodes and 22 computers are equipped with special I/O capabilities. This structure has the advantage to allow massive parallel computations distributed over a high number of nodes. It also benefits the energy consumption, since unused nodes can be suspended to minimize the overall consumption. Figure 1 depicts the layout of the GA144 processor.

There is a wide field of possible applications for the GA144 chip. The low power consumption makes the GA144 predestined for mobile devices and sensors with a limited energy supply as an extended battery life is of high importance in such applications. Moreover, the high computational power opens the field to more complex control systems and research applications. Due to a low price of around 20 USD the GA144 could also be applied in a cluster with multiple chips.

2.3 Programming the Chip

The programming language for the GA144 is a Forth-derivate called *colorForth* in which different colors give different meanings to parts of the program code. For example, comments have their own color, as have addresses and labels. Note

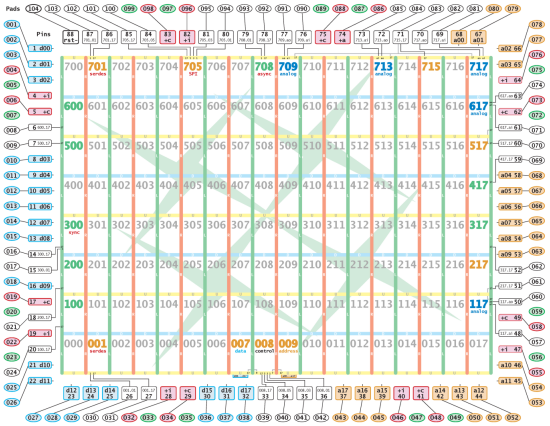


Fig. 1: Layout of the GA144 chip, consisting of an 18x8 array of F18A computing nodes [9].

that Forth itself follows a completely different programming paradigm compared to conventional programming languages such as C or Java and that the coloring is part of the programming as opposed to the syntax highlighting in other languages.

The instruction set for this platform features 32 5-bit opcodes. A 18-bit word consists of three 5-bit slots and one 3-bit slot. Therefore, up to four opcodes can fit into one word. Since the last slot is only 3 bits wide, only eight particular opcodes can be used. These are all the opcodes of which the two least significant bits are zero and therefore are ignored. Figure 2 lists the F18A's instruction set.

Opcode	Notes	Opcode	Notes
;	return	+*	multiply step
ex	execute (swap P and R)	2*	left shift
name ;	jump to *name*	2/	right shift (signed)
name	call to *name*	–	invert all bits
unext	loop within I (decrement R)	+	add (or add with carry)
next	loop to address (decrement R)	and	
if	jump if T=0	or	exclusive or
-if	jump if T≥0	drop	
@p	literal (autoincrement P)	dup	
@+	fetch via A (autoincrement A)	pop	from R to T
@b	fetch via B	over	
@	fetch via A	a	fetch *from* register A
!p	store via P (autoincrement P)	.	nop
!+	store via A (autoincrement A)	push	from T to R
!b	store via B	b!	store *into* register B
!	store via A	a!	store *into* register A

Fig. 2: The instruction set of the F18A processor consisting of 32 5-bit opcodes [8].

The major challenge for a programmer is the strong memory constraint for each node. With only 64 18-bit words in each node larger applications need to be split up and distributed to multiple nodes. This however also offers the opportunity to exploit inherent parallelism in the applications and possibly allow for an improved computation time.

3 Implementing AES on the GA144

The Advanced Encryption Standard (AES) is the most widely used symmetric block cipher and was designed to be efficiently implementable both in hardware and in software. Although it is a well-established block cipher, we briefly review its structure to give rationales for our design choices for the GA144.

3.1 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) block cipher uses a block size of 128 bit and supports key sizes of 128, 192, and 256 bit. During en-/decryption the input block is run multiple times through three different layers. The *Key Addition Layer* and the *Diffusion Layer* apply linear transformations, the non-linear element is the *Byte Substitution Layer* [25].

During *Key Addition* a 128-bit roundkey is XORed to the state. The roundkeys are derived from the main key by a key derivation function.

The *Diffusion Layer* consists of two sub-layers. First, the *ShiftRows Layer* applies a row-wise byte rotation to the state. Afterwards, the bytes are passed to a matrix multiplication in tuples of four bytes within the *MixColumn Layer*.

The *Byte Substitution Layer* substitutes all bytes with their (slightly modified) multiplicative inverse in $\mathbb{F}_{2^8}$ and is usually implemented as table-lookup (S-box) of 256×8-bit entries.

3.2 Design Considerations

Since memory is very scarce in each node of the GA144 it is necessary to spread the tables and the entire encryption over multiple nodes. A natural approach is to split the AES computations by its separate layers. However, while *Key Addition* and *ShiftRows* are simple operations and fit into one node, this does not apply for the *Substitution Layer* and the *Key Schedule*.

We decided to implement the *Substitution Layer* as straightforward 8-bit look-up table. One F18A node can store up to 64 18-bit words. Since the S-box table has 256 8-bit entries, a minimum of $\lceil \frac{256 \cdot 8}{64 \cdot 18} \rceil = \lceil 1.77 \rceil = 2$ nodes are required to store the table. Therefore, we divided the table into two halves and stored two 8-bit entries in one 18-bit word in the program memory. Thus, two nodes are required to store the complete table. The two tables take up the complete program memory of both nodes so that there is no space for additional control. Therefore, a third node is utilized to handle the table lookups appropriately.

The *MixColumn Layer* fits into one node. However, as detailed in the next section, we found that distributing the task over four nodes yields a much better overall performance. Each node mixes one of the state columns using shift and XOR operations to realize polynomial multiplications.

The *Key Schedule* is split over six nodes. One node permanently stores the main key. Another node holds the currently required roundkey and delivers it to the *Key Addition Layer* while a third node schedules the next roundkey. Since the *Substitution Layer* is used when scheduling a roundkey and due to the fact that

the key schedule is computed in parallel to the cipher, a second instance of the *Substitution Layer* needs to be added.

As described in the next section, the overall structure of the cipher is very compact and does not need many control or routing nodes. Only one extra node is needed to realize the I/O and control the implementation.

3.3 Implementation Aspects

The basic layout for the AES encryption does not change for different key lengths. It can be reused when switching to another key length by making minor changes to the layer nodes and some major changes to the key schedule. In total our implementation occupies 17 out of the 144 available nodes.

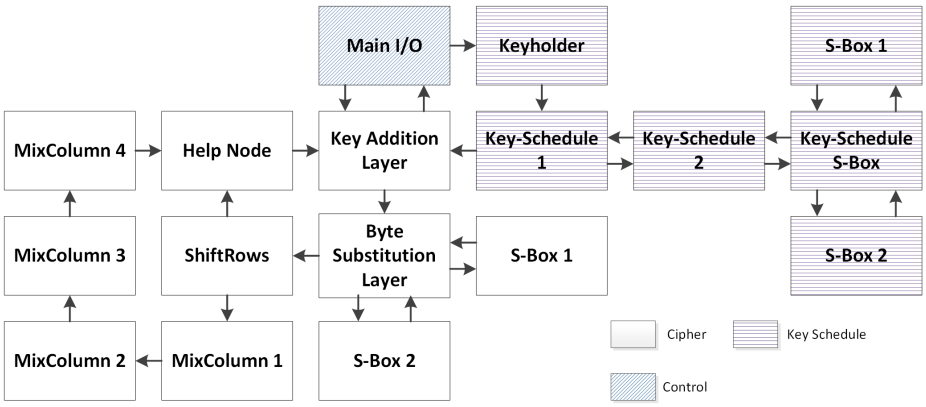


Fig. 3: Data flow diagram of the AES encryption (same for all key sizes).

As shown in Figure 3 the implementation can be divided into three parts: Control, Key Schedule, and Cipher. Each part communicates with the other two throughout the encryption process. The information flow is depicted by the arrows between the nodes, the start-up node being Main I/O. The I/O part is only represented by one node because there is no need for more in our current scenario. This can be easily extended to fulfill more challenging demands. In contrast to the small I/O part, the other two make up the majority of the implementation. The Key Schedule consists of six nodes, while the Cipher occupies nine F18As.

The Main I/O node starts the encryption. It activates the Key Schedule and the Cipher and is also the last node to be active after the encryption is done. Before or after an encryption, Main I/O signals the Key Schedule to reset its roundkeys. Afterwards, the Cipher is started and the plaintext is send to the Key Addition Layer. Then Main I/O waits for the cipher to finish, accepts the ciphertext and stores it for further use. This marks the end of the encryption and all nodes are suspended until the next encryption is started.

The key schedule can be divided into three different parts: the Keyholder, the Key Schedule nodes 1 and 2, and the Key Schedule S-Box. The actual key schedule is performed by the Keyschedule Nodes 1 and 2 with the help of

the KEYSCHEDULE S-BOX. The current roundkey is held in KEY SCHEDULE 1 and is send to the KEY ADDITION LAYER in each round. Since every designated key length has a key schedule which is in some ways a bit different than the other two, the key schedule has to be adjusted if the length of the key changes.

The cipher itself consists of three different layers as explained before, which are implemented in nine nodes. The HELP NODE'S task is to either forward the result of the *MixColumn Layer* to KEY ADDITION LAYER or skip the *MixColumn Layer* in the last round by redirecting the output of SHIFTROWS to the KEY ADDITION LAYER.

Through simulation we acquired performance figures for the separate layers and compared them among each other to find possible bottlenecks. In the first version of our implementation we realized the *MixColumn Layer* in one node. It turned out that it consumed 40% more time than second slowest layer which was the *Byte Substitution Layer*. Therefore, we decided to distribute the *MixColumn Layer* over four nodes to improve its performance. This helped to reduce the overall time consumption of the layer by 70%.

Our AES decryption implementation uses a similar layout as the encryption. A main difference is the necessary precomputation of the roundkeys. This procedure needs additional nodes and raises the total number of nodes from 17 to 21. The main differences between decryption and encryption are a reorganized key schedule and a added possibility to store the roundkeys. Furthermore, the layers are inverted and the order of them is changed as well.

3.3.1 Multiple Instances

Since only 17 out of 144 nodes are used for one AES encryption unit, it is possible to place multiple instances on one chip and run them simultaneously. However, there are a few constraints on how and where to place these instances. For a start, each instance has to have a connection to external I/O nodes of the GA144 to be able to communicate with the outside world. Furthermore, the internal structure of the nodes obviously needs to be preserved.

A possible layout with six instances of the AES encryption unit is shown in Figure 4. Some instances need extra nodes to reach the I/O nodes caused by constraints of the chip's layout, so a total of 112 nodes instead of $17 \cdot 6 = 102$ are occupied in this design.

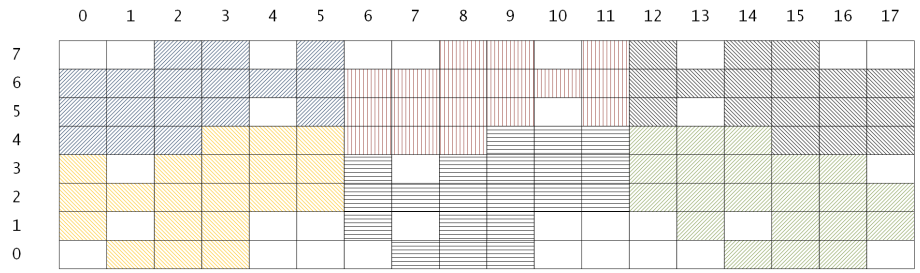


Fig. 4: Distribution of six AES encryption instances on a GA144 chip.

In case of multiple decryption instances one has to keep in mind that the extended key schedule requires more nodes for the decryption unit. Thus, the same layout cannot be reused. Again, the same constraints as in case of the encryption unit hold. A possible layout of five decryption instances on the GA144 occupies 120 nodes. This is one instance less than in case of encryption but had to be expected. Moreover, the same technique as before was applied to connect the Main I/O node of each instance with the external I/O of the chip to ensure functionality.

3.3.2 Extended Byte Substitution Layer

After the improvement of the *MixColumn Layer*, the *Byte Substitution Layer* of AES is the implementation's bottleneck. In order to further improve the overall performance, the *Byte Substitution Layer* is instantiated twice. With two instances running in parallel, the time consumption of the layer is halved since the state bytes can be substituted independently of each other. Figure 5 shows a possibility to implement a second *Byte Substitution Layer* with four additional nodes, raising the total number of nodes to 21.

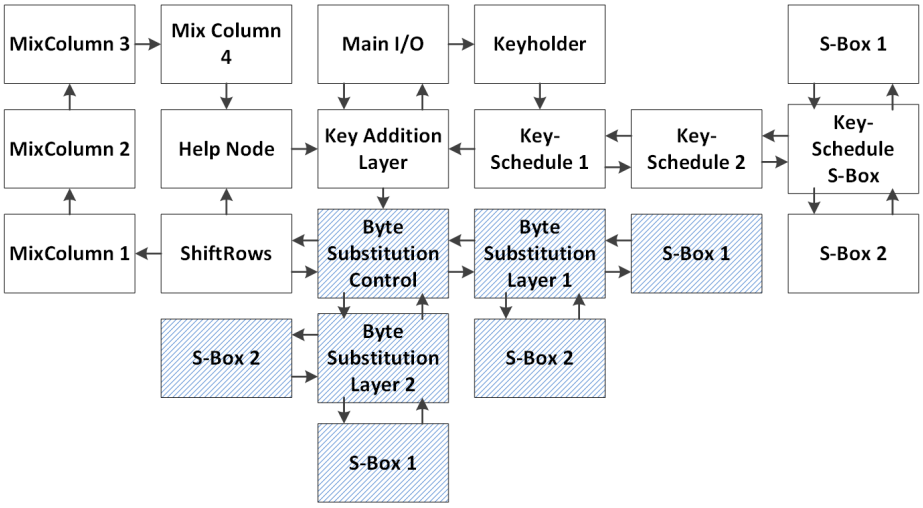


Fig. 5: Data flow diagram of an AES encryption with extended Byte Substitution Layer.

The Byte Substitution Control receives its input from Key Addition Layer. Incoming bytes are distributed equally to the Byte Substitution Layer Nodes 1 and 2. Each node substitutes eight bytes per round and returns the output to Byte Substitution Control which forwards the result to ShiftRows. From this point on, the design is equal to the one described in Sec. 3.3.

To increase the performance of the encryption even further more parallel instances can be added. Note that in the new design the *MixColumn Layer* consumes the most time. Therefore, it is more beneficial to add an additional instance of the

MixColumn Layer than another *Byte Substitution Layer*. The limit of this time-area-tradeoff is reached with four instances of the *MixColumn Layer* and 16 instances of the *Byte Substitution Layer*. In such an implementation, all input bytes would be processed in parallel. However, routing overhead will complicate this approach.

3.3.3 Pipelined Implementation

Another approach to increase performance is to pipeline the implementation. Hence, we unroll the AES encryption and implement each round separately. This method increases performance if multiple blocks are encrypted and comes at the cost of higher resource consumption. Figure 6 shows the structure of one round of the pipeline.

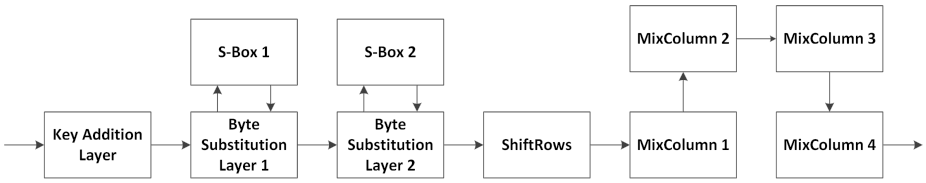


Fig. 6: One round of the pipelined AES encryption.

The Key Addition Layer is similar to the standard implementation. It adds the current roundkey to the state of the cipher. Instead of receiving the roundkey from the key schedule, the key of the corresponding round is stored permanently. Therefore, the key of the pipelined AES encryption is fixed and cannot be changed during runtime. It also implies that the roundkeys need to be computed before compilation.

After the key is added to the state, the Key Addition Layer sends the new state to the *Byte Substitution Layer*. The *Byte Substitution Layer* consists of four nodes with S-Box Nodes 1 and 2 being the same as in the standard implementation. The main node is split up over two nodes to allow a structure that fits into two rows. It also increases the speed of our implementation, since both Byte Substitution Layer nodes can work in parallel. Byte Substitution Layer 1 performs all substitutions that need S-Box 1. All other inputs are directly forwarded to Byte Substitution Layer 2 which uses S-Box 2.

The substituted values are directly forwarded to ShiftRows. This node is the same as in the standard implementation. It stores and reorders the input bytes.

After completion, the state is fed to the four *MixColumn* nodes. Each node computes all four values of one column. They operate in parallel and forward the result to the next round.

Figure 7 depicts a completely pipelined implementation of the AES-256 encryption using 140 nodes. The implementation of AES-128 and AES-192 are similar with fewer rounds. The encryption starts at the upper left corner and runs over the whole chip to the lower left corner. The design of the decryption is the same with inverted layers and a reverse order of roundkeys.

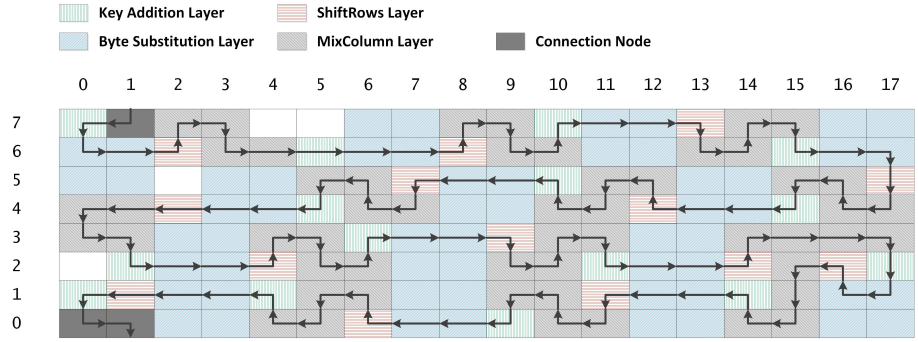


Fig. 7: A pipelined implementation of the AES-256 encryption on the GA144.

The main advantage of the unrolled structure is the possibility of massively parallel computation. Several plaintext blocks can be encrypted before the previous ones are finished. This is feasible since every round has its own nodes and does not depend on the same resources as the previous or next round. Our design does not improve the time consumption of a single encryption, but excels when a large number of blocks is consecutively en-/decrypted. The first block will arrive with latency comparable to a non-pipelined implementation, but all following blocks will be processed much faster. This is possible since their computation can start immediately after the prior block has left the first stage.

A disadvantage are the fixed roundkeys, which require the chip to be reprogrammed if the key changes. Moreover, a different I/O setting has to be taken into consideration. The input to the cipher will be given from one side, iterate through the whole chip and will be available on the other side.

4 Implementing RSA on the GA144

The RSA cryptosystem is still one of the most widely used asymmetric cryptographic scheme. Its security is based on the integer factorization problem which is believed to be hard to solve for appropriately chosen parameters. In the following we briefly summarize RSA and give design rationales for the GA144.

4.1 The RSA Asymmetric Encryption/Signature Scheme

The main operation used in RSA is exponentiation in the integer ring $\mathbb{Z}_m$ where $m = p \cdot q$ represents the modulus. The public key $pk = (e, m)$ consists of an exponent e and a modulus m. The secret key $sk = (d, p, q)$ consists of the inverse of the encryption exponent $d = e^{-1} \ mod \ \phi(m)$ and the prime factors p and q. For secure applications the parameters are at least 1024-bit in size which makes the modular exponentiation operations very complex.

To encrypt a plaintext g the sender computes $c = g^e \ mod \ m$ using the receivers e and m. The receiver decrypts c by calculating $g = c^d \mod m$. Digital signatures can be created using the same procedure by switching the roles of the exponents.

4.2 Design Considerations

In contrast to AES, RSA does not iterate over a series of layers but uses modular arithmetic with large parameters (≥ 1024 bits). This again is a particular problem due to the scarce memory in each node. For our design, we chose to use RSA-1024 with parameters of length 1024 bit. Thus, one RSA parameter fits exactly into the memory of one F18A node.

For exponentiation we use the standard square-and-multiply algorithm [21] in combination with a word-based version of the Montgomery multiplication algorithm [21] which works with words of 16 bits. The core multiplication is implemented in serial-parallel fashion, i.e., multiplying 1024 bits with one word. The multiplier itself is implemented as systolic array [16]. Figure 8 shows a subset of the basic multiplication structure.

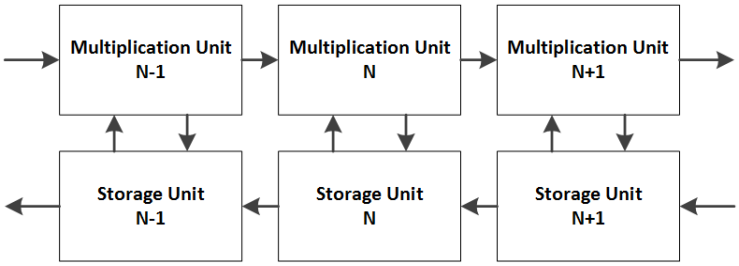


Fig. 8: An example extract of the systolic multiplication array.

The systolic array consists of MULTIPLICATION and STORAGE UNIT pairs. They interact with each other to compute their corresponding results by propagating carries. The MULTIPLICATION UNITS have the task of calculating a multiplication of a one-word factor with multiple words supplied from the STORAGE UNITS. Each of the STORAGE UNITS holds three words of the multiplier, of the intermediate result, and of the modulus m. During the steps of the *Montgomery Product*, they send the required parts to their corresponding MULTIPLICATION UNIT, receive the result and store it. Moreover, they operate independently of the MULTIPLICATION UNITS at the end of the product by shifting the final result one word to the right.

At the beginning of the multiplication the factor is fed to the MULTIPLICATION UNITS from the left. It is copied and forwarded through the whole array so that all units receive a copy. Then the actual multiplication is computed in parallel in all nodes and afterwards the carries are exchanged with the next MULTIPLICATION UNIT in the array. This has the advantage of speeding up the process by allowing parallelism. For RSA-1024 a total of 22 storage units is required since

$$22 \cdot 3 \; words \cdot 16 \; \tfrac{\text{bits}}{\text{word}} = \; 1056 \; bits > 1024 \; bits.$$

Note, intermediate results during the *Montgomery product* cannot exceed 1048 bit at any moment, thus overflows during the multiplication do not occur.

4.3 Implementation Aspects

Our RSA-1024 implementation takes up 107 nodes on the GA144. As mentioned before it consists of several different elements that are combined to form the complete cipher. The *Montgomery multiplication* is the largest component and occupies 55 nodes, whereas the 1024-bit modulus reduction requires 15 nodes. The remaining nodes are used to implement the MAIN CONTROL, storage nodes for the parameters, an extended Euclidean algorithm, I/O, and connection nodes.

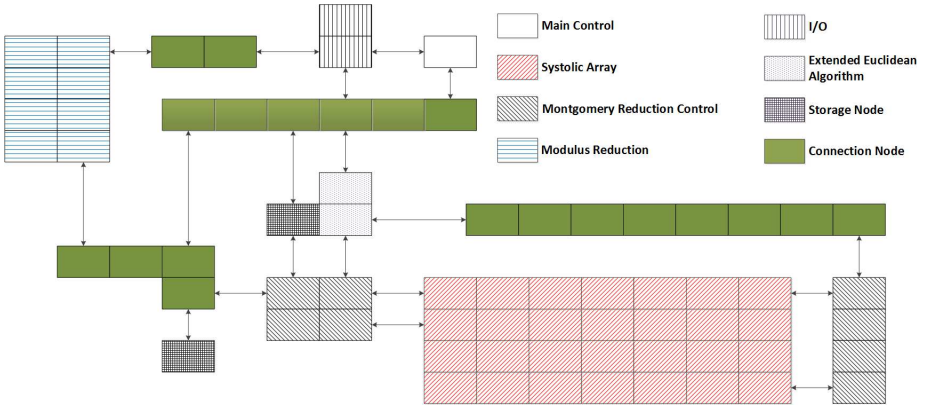


Fig. 9: Simplified data flow diagram of our RSA implementation.

Figure 9 depicts a simplified diagram of our implementation. It shows the connections between the fundamental parts of the encryption. The parts are reduced and do not consist of the actual number of occupied nodes. Every process is controlled by the MAIN CONTROL node at the top which is also the last node to be active after the computation is finished.

The RSA parameters can be set from the outside using the I/O CONTROL node which forwards the parameters to the desired storage nodes. Additionally, the result of the exponentiation can be made accessible from the outside by transferring it to the I/O MEMORY. This procedure is utilized after the RSA operation is finished in order to retrieve the output.

The task of the MAIN CONTROL unit is to send out commands to the surrounding nodes. It is accessed through an asynchronous connection and supplies the outside with simple functions to control the flow of the cipher. Especially the I/O CONTROL node is activated by it to set the parameters of the exponentiation.

Since the RSA implementation takes up more than half of the available nodes on the GA144, instantiating multiple units as shown for AES is not feasible. We like to close this section with some further ideas that have not been implemented yet. The Sliding Window algorithm [21] is a more efficient way for modular exponentiation and splits the exponent into small windows and uses precomputed values to speed up the exponentiation process. This requires additional nodes to store the results of the precomputation but is likely to increase performance. Additionally, the Chinese Remainder Theorem (CRT) can be used if the prime factors

p and q of modulus m are known. The CRT splits the exponentiation into two separate instances based on the prime factors of the modulus which are only half the size of m. The computational complexity is reduced by a factor of nearly 4 at the drawback of enabling a potential fault injection attack that can compromise the secret key.

5 Results

In the following we present our results and measurements of the execution time and power consumption and compare them to other published results on similar platforms.

We measure the power consumption and execution time using a digital oscilloscope (PicoScope 5203 [28]) with two channels. *Channel A* is set as trigger and *Channel B* is used to measure the voltage over a 18 Ω resistor inserted into the ground path. Immediately before the encryption starts the trigger pin is set and held until the end of the encryption. Thus, the computation time of the cryptosystems can be extracted by taking the time between a rising and a falling edge of the trigger signal. In addition, we extract the power consumption of the chip from *Channel B*, which we use in combination with the execution time to compute the overall energy consumption.

Note that the performance of an asynchronous processor is influenced by its supply voltage. Thus, we measure and analyze the implementations for three different supply voltages (1.8 V, 2.0 V, 2.2 V).

5.1 Results for AES

For each supply voltage we average our measurements over 10,000 AES en-/decryptions. The keys and plaintexts change after every encryption and the resulting ciphertext is used as source of randomness for the next key and plaintext. Our results are summarized in Table 1.

Table 1: Execution time and energy consumption of standard AES-128,-192,-256 en-/decryption on the GA144.

Supply voltage	AES mode	Encryption			Decryption		
	Key size [bit]	128	192	256	128	192	256
1.8 V	Time [μs]	45.1	51.4	64.1	74.2	85.2	110.6
	Energy [μJ]	0.63	0.65	0.86	0.97	1.14	1.53
2.0 V	Time [μs]	41	49.6	56	72.4	83.3	100.2
	Energy [μJ]	0.77	0.74	1.078	1.36	1.57	1.86
2.2 V	Time [μs]	37.9	44.3	52.7	66.6	76.7	93.7
	Energy [μJ]	0.90	0.92	1.26	1.35	1.60	2.19

AES encryption with 128-bit keys is the fastest and most energy efficient operation mode. This is due to the fact that this variant executes the smallest number

of rounds and thus needs less operations to complete. In comparison to encryption, decryption takes more time and energy which can be explained by the required roundkey pre-computation and the additional computations for the inverse layers.

With a rising supply voltage, the execution time of all modes decreases while the overall energy consumption rises. Interestingly, the energy consumption only slightly differs between the 128-bit and the 192-bit encryption. At a supply voltage of 2.0 V the 192-bit mode even outperforms the 128-bit mode in terms of energy-efficiency.

To quantify the performance gained by using the extended *Byte Substitution Layer* we use simulation results and compare them to the standard AES implementation. The same 1000 plaintexts (uniformly selected at random) are encrypted and the average computation time is measured for all three key sizes for the standard implementation and the implementation with the extended Byte Substitution Layer. As shown in Table 2 the extension of the *Byte Substitution Layer* yields a consistent improvement of the execution time of 21% for all key sizes. Furthermore, Table 2 verifies the quality of the simulation as the simulated time consumptions for standard AES are close to the measured values for 1.8 V in Table 1.

Table 2: Simulated execution times of the standard AES implementation and the AES implementation with the extended Byte Substitution Layer.

Key size [bit]	128	192	256
Time - standard [μs]	45.5	54.7	63.2
Time - extended [μs]	35.6	43.0	49.5
Improvement [%]	21.6	21.3	21.6

For comparing the pipelined with the standard AES implementation we use a similar approach. We simulate the pipelined implementation for 1000 plaintexts and compare its throughput to the standard implementation. Since the pipeline uses the whole chip and the standard version fits into 17 nodes, we use the model with multiple instances on one chip described in Sec. 3.3.1. Therefore, we use the simulated values from Sec. 3.3.2 and multiply the throughput of the standard implementation with six. Unfortunately, the simulator is not capable of handling more than 138 nodes. Therefore, we estimate the throughput for the pipelined AES-256 encryption using the throughputs for the other two key sizes as a basis. The coefficient between the 128 bit and the 192 bit version is expected to be the same as the coefficient between the 192 bit and the 256 bit version, since in both cases simply two rounds are added to the implementation. Differences in the key schedule do not matter since the round keys are stored instead of being computed on-the-fly.

Table 3 shows the results of our simulations. For all key sizes the pipelined implementation increased the throughput between 168% and 228%. Since the main advantage of a pipelined implementation comes with the processing of a large number of inputs, the performance will decrease if only few blocks are encrypted. The standard design is also more flexible and allows different keys for the different instances.

Table 3: Simulated throughput comparison of the standard and the pipelined AES encryption on the GA144. *Estimation based on the throughputs of pipelined AES-128 and AES-192.

Key size [bit]	128	192	256
Throughput - standard [$Mbit/s$]	16.88	14.04	12.15
Throughput - pipeline [$Mbit/s$]	45.35	42.57	39.96*
Improvement [%]	168.66	203.20	228.88

5.2 Results for RSA

The RSA implementation is measured in a similar manner, averaging over 1,000 RSA operations. We change the modulus after every operation and the exponent and the message after every second operation. Note that RSA encryption and decryption use the same exponentiation unit. Hence, their runtimes are similar and only depend on the chosen parameters. Furthermore, we decided to select full-size random parameters for a fair comparison and do not use short exponents to speed up the encryption. Our results are presented in Table 4.

Table 4: Execution time and energy consumption of RSA en-/decryption on the GA144.

Supply voltage	1.8 V	2.0 V	2.2 V
Time [ms]	598.1	513.7	462.9
Energy [mJ]	22.3	24.6	27.9

Again, with a rising supply voltage the execution time is decreased at the cost of an increased energy consumption.

5.3 Comparison

In the following we compare the time and energy consumption of our implementations of AES and RSA to implementations on similar embedded platforms.

Since one field of application are portable devices and remote sensing, we compare our standard AES implementation with an implementation on a *MicaZ* sensor node for a wireless sensor network (WSN). It is a widely-used 2.4 GHz board equipped with an 8-bit Atmel AVR ATmega128L microcontroller clocked at 8 MHz and is especially designed for low-power applications. In [35], a AES software implementation as well as a hardware-supported AES implementation (using the Chipcon CC2420 RF transceiver chip) are evaluated on this platform.

A comprehensive survey of the performance and energy-efficiency of several block ciphers (among them AES) on an inexpensive, low-power Atmel AVR ATtiny45 8-bit microcontroller is given in [5].

In [13] and [14], ASIC implementations of AES aiming for a low power consumption in WSN and RFID applications are presented. We include these results to show the gap between microcontrollers and special-purpose hardware implementations.

Additionally, we compare our implementation with a recent AES implementation [18] for a AsAP many-core processor array prototype that offers 167 cores [31].

Since the above mentioned implementations focus on AES encryption with a 128-bit key, we compare them against our single-instance AES-128 encryption at a supply voltage of 2.2 V. The comparison is given in Table 5.

Table 5: Comparison of the time and energy consumption of AES-128 encryption on embedded platforms.

Platform	Time [μs]	Energy [μJ]
GA144 (0.18 μm)	37.9	0.90
AVR ATtiny45 [5]	455.7	19.2
MicaZ software [35]	885.8	19.16
MicaZ HW-assisted [35]	350.6	26.82
Many-core prototype (65 nm) [18]	0.126	0.198
ASIC (0.13 μm) [14]	2140	0.051
Low-power ASIC (0.13 μm) [13]	1.23	0.005

The AES implementations on the ATtiny45 and the MicaZ microcontroller are clearly outperformed by our implementation. In terms of speed, AES encryption runs 12-23 times faster on the GA144. Even when compared to a hardware-accelerated AES available on the MicaZ board, our implementation still executes a AES encryption more than 9 times faster. Note that in this comparison we restrict ourselves to a single instance of AES. As stated before, up to six instances of the AES-128 encryption can run in *parallel* on a single GA144 which would further increase the GA144's advantage.

Moreover, the energy consumption of AES on the GA144 is 21 times less compared to AES software implementations on the microcontrollers. Using the Chipcon CC2420 hardware-accelerated AES even consumes 30 times more energy than the GA144.

A recent AES implementation for the AsAP many-core processor array prototype occupies 39 out of 164 programmable cores. The processor is fabricated using a considerably smaller CMOS technology (180 vs. 65 nm) and its cores appear to be more powerful than the F18As. It seems to be a promising architecture with fast execution times and a low energy consumption with the drawback of only being a prototype as compared to the freely available GA144. Note that the AsAP result are reported for a fixed key implementation.

When comparing our results to special-purpose hardware designs it becomes clear, that even with a low-power processor and carefully optimized implementations, the performance and energy consumption can not keep up. The advantages of an ASIC naturally come at the cost of much less flexibility, higher development costs and longer development cycles.

RSA implementations have been reported for similar microcontroller platforms as in the case of AES. In [12] and [33], RSA implementations for the Mica2 and

Table 6: Comparison of the time and energy consumption of RSA encryption on embedded platforms. *The energy consumption is linearly interpolated based on the Mica2 results.

Implementation	Time $[ms]$	Energy $[mJ]$
GA144 (0.18 μm)	513.7	24.6
Mica2 [12] [32]	10,990	304
MicaZ [33]	7,900	218.5*
AVR ATmega128 [19]	4,730	283.7*
DSRCP ASIC (0.25 μm) [7]	32.1	2.41
RCP ASIC (0.13 μm) [3]	3.84	1.75

the MicaZ WSN platform are presented. Furthermore, a RSA implementation for an AVR ATmega128 is given in [19]. Unfortunately, the energy consumption is only reported for the implementation on the Mica2 platform. Since the same microcontroller is used by all implementations, we estimate the energy consumption of [19] and [33] by linear interpolation taking into account the different execution time and clock frequency.

Special-purpose reconfigurable cryptographic processors (RCP) are the counterpart to the microcontroller approach. Explorations on how to design an energy-efficient ASIC capable of performing RSA operations are given in [3] and [7].

All of the above mentioned implementations run RSA-1024, so they are a good fit for comparison. Table 6 lists time and energy consumption of our RSA implementation at a supply voltage of 2.0 V and the reported results on other embedded platforms.

A similar behavior as in the case of AES can be observed when comparing RSA on a GA144 to other microcontroller platforms. An RSA operation is performed 9 - 21 times faster and consumes 12 times less energy at the same time. Specially designed ASIC implementations of RSA however are much faster and consume around a tenth of the energy.

Overall the GA144 chip turns out to be a good choice for WSN and RFID applications when compared to other low-power microprocessor for both ciphers, since it operates faster and is more energy efficient at the same time. Its performance and energy consumption lies in between common microcontrollers and special-purpose ASICs.

6 Side-Channel Analysis of the AES Implementation

In this section, we analyze the basic implementation of the AES presented in Section 3 from an adversary's perspective. Since the AES is secure with respect to analytical attacks, we target the actual implementation, employing side-channel attacks as briefly introduced in Section 1. To this end, we first recall the technique of Correlation Power Analysis (CPA) in Section 6.1, followed by a description of the utilized measurement setup (Section 6.2) and the achieved results (Section 6.3).

6.1 Correlation Power Analysis

CPA [1] (and the predecessor DPA [15]) are based on the evaluation of many side-channel measurements with varying input data for the targeted algorithm. Then, a brute-force attack with additional information is performed on a part of the algorithm. For instance, an attack on the AES usually only targets the first round, for which each S-Box is processed separately. Hence, each subkey entering one specific S-Box can be recovered independently from the remaining key bits. By testing all possible candidates for a given subkey (e.g., $2^8 = 256$ for the AES), the full key can be extracted in a divide-and-conquer manner.

The key idea behind CPA and related methods is that side-channel information can be used to identify the correct subkey amongst the candidates. A statistical test is employed to determine which subkey candidate is most likely to have caused the observed leakage. This is where according evaluation methods differ. For example, DPA is based on computing the difference between two measurement sets, while CPA uses the correlation coefficient [1] to perform this test for dependency.

The process of performing a CPA can be divided into two steps. In the measurement phase, the adversary has physical access to the device and records some side-channel signal (e.g., the power consumption or the EM emanation during the cryptographic computation) that is related to the processed data. This step is repeated N times with varying input data M_i, yielding N time-discrete waveforms (usually called *traces*) $x_i(t)$ with T points each.

In the evaluation phase, the key is recovered by fixing a (small) subset $\mathcal{K}_{cand} \subseteq \mathcal{K}$ and considering all key candidates $k \in \mathcal{K}_{cand}$: for each $k \in \mathcal{K}_{cand}$ and for each $i \in \{0, \ldots, N-1\}$, a hypothesis $V_{k,i}$ on the value of some intermediate is computed. This hypothesis could, for example, be the output of one AES S-Box in the first round.

Using a power model f, the hypothesis $V_{k,i}$ on the value of some intermediate value is then mapped to $h_{k,i} = f\left(V_{k,i}\right)$ to describe the process that causes the side-channel leakage. In practice, the power model is often either the Hamming Weight (HW) or Hamming Distance (HD) model [20].

Both $h_{k,i}$ and $x_i(t)$ are treated as observations of discrete random variables. In order to detect the dependency between $h_{k,i}$ and $x_i(t)$, the correlation coefficient $\rho_k(t)$ (for each point in time $t \in \{0, \ldots, T-1\}$ and each key candidate $k \in \mathcal{K}_{cand}$) is given by Equation 1:

$$\rho_k(t) = \frac{cov\left(x(t), h_k\right)}{\sqrt{var\left(x(t)\right) var\left(h_k\right)}} \tag{1}$$

with $var(\cdot)$ indicating the sample variance and $cov(\cdot, \cdot)$ the sample covariance according to the standard definitions [34]. Finally, the key candidate $\hat{k}$ with the maximum correlation $\hat{k} = \arg\max_{k,t} \rho_k(t)$ is assumed to be the secret key k^{device} used by the device.

6.2 Measurement Setup

For acquiring the side-channel signal of the GA144, we initially attempted to utilize the power consumption of the device measured over the shunt resistor in the ground path mentioned in Section 5. However, the achievable signal amplitude was

relatively low (approximately 10 mV at a minimum input range of the Picoscope of ±100 mV), and further increasing the value of the 18 Ω resistor resulted in unstable operation of the GA144.

Hence, we opted for recording the EM emanation of the device. This approach has the additional advantage that it also works in cases where the supply lines are not easily accessible or where additional ICs cause noise on the power supply. For the GA144, we placed a near-field probe made by Langer EMV [17] on various positions on and around the GA144 IC to experimentally determine the position with the maximum signal amplitude. We found that the side-channel leakage is highest when placing the probe on the decoupling capacitor that buffers the supply voltage of the GA144. The measurement setup is depicted in Figure 10.

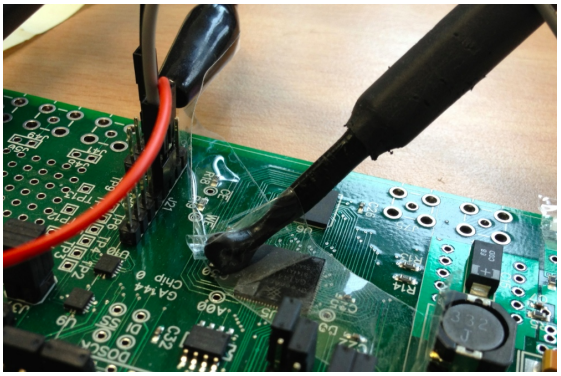


Fig. 10: Measurement setup for side-channel analysis of the GA144. EM probe on decoupling capacitor which buffers the supply voltage.

Using the determined probe position, we recorded 5,000 traces (together with the corresponding, uniformly distibuted AES plaintexts) during the execution of the AES using the PicoScope. The sample rate was set to $f_s = 500$ MHz. The supply voltage of the GA144 was set to 2 V. As the AES key for our experiments, we used the value `2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c`[1] taken from the FIPS specification [25].

6.3 Results

Figure 11 exemplarily depicts an EM trace during the execution of the AES. The ten rounds of the AES can be identified as "humps" (nine full rounds and final round without `MixColumns`) in the trace. Moreover, as expected, the overall EM emanation (and hence the power consumption) of the GA144 significantly increases during the AES operation.

Before performing the side-channel attack, we noted that the acquired EM traces are *misaligned* after a certain point of time (with respect to the begin of the

[1] Note that using a known key for profiling a device is a standard practice in side-channel research

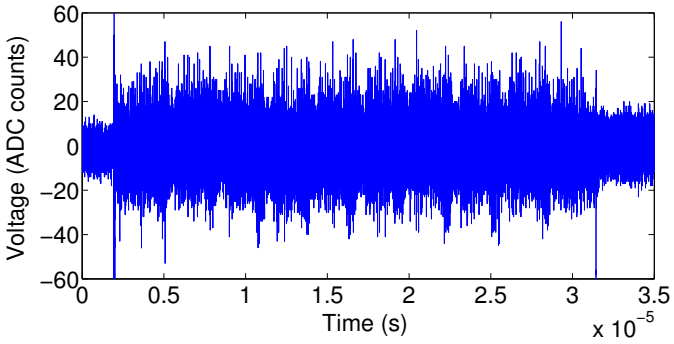
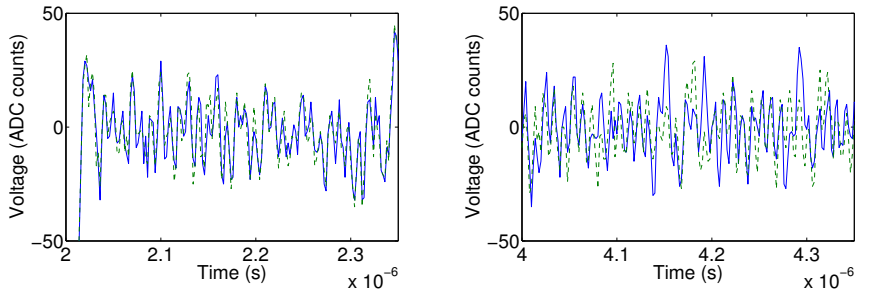


Fig. 11: EM trace for the execution of the AES on the GA144, $f_s = 500\,\text{MHz}$.

AES operation), i.e., for different traces, a specific operation within the algorithm is executed at different points in time. This observation is illustrated in Figure 12 for two traces (first: blue, solid, second: green, dashed): While the traces are perfectly aligned at the beginning (Figure 12a), the timing varies significantly already within the first round (Figure 12b).



(a) Beginning ($2\,\mu$s), aligned (b) Within first round ($4\,\mu$s), misaligned

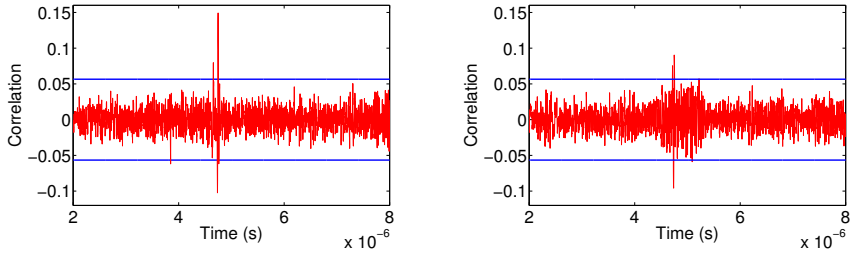
Fig. 12: Temporal (mis)alignment in different parts of the EM traces.

Note that this misalignment is mainly not caused by the asynchronous nature of the GA144. Rather, as the primary reason for this behavior, the timing of the *Byte Substitution Layer* is data-dependent[2] and hence causes the observed de-synchronization for different input values. To overcome this problem, we re-aligned the traces using a standard least-squares pattern matching approach (cf. for instance [20]), iterating over a range of reference patterns.

For the CPA, we focused on the first round of the AES (at approximately $2 \ldots 6\,\mu$s) and thus predict the output of each S-Box in the first `SubBytes` layer. As the power model, we used the HW, i.e., $h_{k,\,i} = \text{HW}(\text{S-Box}(M_i \oplus (\hat{K} = k)))$, where M_i is the input byte entering the targeted S-Box and $\hat{K}$ the respective key byte.

[2] This characteristic could likely also be used to carry out a timing attack, however, we did not further investigate this issue for the purposes of the present paper.

We then carried out a CPA for each S-Box and for each reference pattern using 5,000 traces in order to determine the alignment pattern for each S-Box that maximizes the correlation coefficient for the correct key candidate. The resulting correlation coefficient (for the relevant part of the trace) is exemplarily depicted for S-Box 1 and 16 in Figure 13. The vertical blue lines indicate the expected "noise level" of $4/\sqrt{\text{No. of traces}}$ [20], yielding a value of approximately 0.057 for 5,000 traces. For both S-Boxes, the correlation clearly exceeds this boundary, with a maximum (absolute) value of 0.149 (for S-Box 1) and 0.096 (for S-Box 16), respectively.



(a) S-Box 1 (b) S-Box 16

Fig. 13: Correlation coefficients for correct key candidate for S-Box 1 and 16 after 5,000 traces (with best alignment for each S-Box).

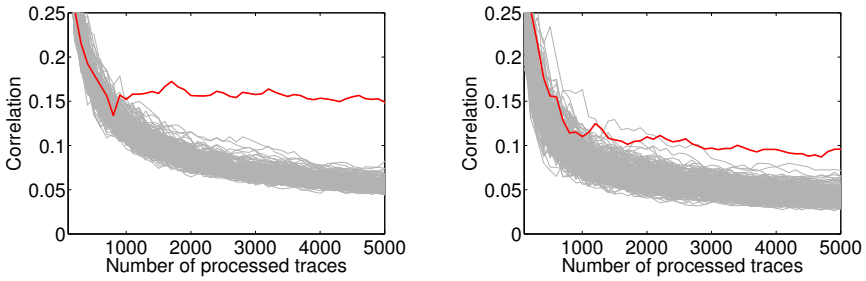
For different S-Boxes, we observed different values for the maximum correlation. For this reason, S-Box 1 and S-Box 16 were chosen as examples because they represent these cases. More precisely, it turned out that for S-Boxes 4, 8, 12, and 16, the correlation coefficient was lower than for the other S-Boxes, with values of 0.086, 0.084, 0.077, and 0.096, respectively.

This effect is likely caused by the implementation details of the AES, e.g., that these values are "touched" by less instructions and hence the quality of the alignment reduces. Nevertheless, for all S-Boxes, the still relatively high correlations allow to identify the correct key candidate with less than 5,000 traces. To this end, Figure 14 depicts the maximum correlation for the correct key candidate (red) and all wrong candidates (gray) over the number of processed traces for S-Box 1 and 16.

Evidently, the correct key for S-Box 1 can be identified with approximately 1,000 traces, while for S-Box 16 approximately 3,000 traces are needed. Alternatively, one can also estimate the minimum number of required traces as $28/\rho^2$ [20] (with ρ the correlation for the correct key), resulting in 1,261 traces (S-Box 1) and 3,038 traces.

For the "worst" S-Box 12 (with $\rho = 0.077$), this yields an overall minimum number of traces of 4,722. As a side note, an adversary could reduce this number by performing an exhaustive search (with complexity 2^{32}) for the keys of S-Box 4, 8, 12, and 16. This would lead to a minimum number of approximately 2,700 traces for extracting the full 128-bit AES key.

To summarize the results of this section, we conclude that the GA144 is— as expected for an unprotected embedded device— susceptible to side-channel



(a) S-Box 1 (b) S-Box 16

Fig. 14: Evolution of the maximum correlation for S-Box 1 and 16 over the number of processed traces. Red: correct key candidate, gray: wrong key candidates.

attacks. Using less than 5,000 EM traces, an adversary is able to extract the full 128-bit key of our AES implementation. With the presented measurement setup, 5,000 traces could be acquired within 6 min, i.e., at a rate of 13.89 $^{\text{traces}}/_{\text{s}}$.

For real-world applications, the inclusion of suitable countermeasures is mandatory to prevent power attacks as shown above. Here, both hiding and masking techniques could be taken into account, cf. [20] for a summary of possible approaches. In particular, an interesting option could be to examine if the asynchronous logic style of the GA144 can be used to realize timing-based countermeasures (e.g., random delays) with a very low overhead. For instance, subtle timing variations due to varying environmental conditions may serve (with suitable post-processing) as an intrinsic entropy source. However, we did not thoroughly examine the behavior of the GA144 in this regard and hence leave this point for future work.

Timing-based countermeasures could be combined with algorithmic modifications like masking [2, 4, 29] to further increase the level of protection with respect to side-channel attacks. As mentioned in Sect. 3.3, the basic AES implementation uses 17 of 144 available nodes. Given the typical overhead caused by masking schemes (e.g. doubled memory requirements and three times the code size [29]), it is thus likely that a protected implementation would fit the resources available on the GA144. It would hence be interesting for subsequent studies to practically implement a masked AES on the device to determine the actual performance and size requirements.

7 Conclusion

In this paper we presented novel implementations of AES and RSA on the GA144 multi-core processor. We explored several design strategies and showed that it is possible to implement even very computationally expensive ciphers efficiently.

When comparing performance and energy consumption to similar implementations on other devices, it becomes evident that our implementations are very competitive and are able to outperform their counterparts on other low-power devices. Hence, the GA144 is a good candidate for a powerful sensor node operating at a low energy consumption.

With respect to implementation attacks, we found that the GA144 can, similar to other unprotected devices like microcontrollers and FPGAs, be attacked by means of side-channel analysis. For the (unprotected) AES implementation, less than 5,000 EM traces were sufficient to extract the full 128-bit key. The fact that the device employs asynchronous logic appears to offer little to no additional protection in this regard. Hence, designers of cryptographic algorithms for the GA144 should take the threat of side-channel attacks into account and implement suitable countermeasures.

Acknowledgment

This work was supported in part by grant 01ME12025 SecMobil of the German Federal Ministry of Economics and Technology.

References

1. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES'04*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
2. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
3. J.-H. Chen, M.-D. Shieh, and W.-C. Lin. A high-performance unified-field reconfigurable cryptographic processor. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(8):1145–1158, 2010.
4. J.-S. Coron and L. Goubin. On Boolean and Arithmetic Masking against Differential Power Analysis. In c. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'00*, volume 1965 of *LNCS*, pages 231–237. Springer, 2000.
5. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. *Progress in Cryptology-AFRICACRYPT 2012*, pages 172–187, 2012.
6. J. J. A. Fournier, S. W. Moore, H. Li, R. D. Mullins, and G. S. Taylor. Security evaluation of asynchronous circuits. In C. D. Walter, Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2003.
7. J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *Solid-State Circuits, IEEE Journal of*, 36(11):1808–1820, 2001.
8. GreenArrays. DB001 - F18A Technology Reference. `http://www.greenarraychips.com/home/documents/greg/DB001-110412-F18A.pdf`, as of March 24, 2021.
9. GreenArrays. DB002 - G144A12 Chip Reference. `http://www.greenarraychips.com/home/documents/greg/DB002-110705-G144A12.pdf`, as of March 24, 2021.
10. GreenArrays. PB003 - F18A Computers. `http://www.greenarraychips.com/home/documents/greg/PB003-110412-F18A.pdf`, as of March 24, 2021.
11. GreenArrays. PB004 - F18A I/O and Peripherals. `http://www.greenarraychips.com/home/documents/greg/PB004-110412-F18A-IO.pdf`, as of March 24, 2021.
12. N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 925–943, 2004.
13. P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen. Design and implementation of low-area and low-power AES encryption hardware core. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 577–583. IEEE, 2006.
14. J.-P. Kaps and B. Sunar. Energy comparison of AES and SHA-1 for ubiquitous computing. *Emerging Directions in Embedded and Ubiquitous Computing*, pages 372–381, 2006.

15. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99*, LNCS, pages 388–397. Springer, 1999.
16. H. Kung and C. Leiserson. Algorithms for VLSI processor arrays. *Introduction to VLSI systems*, pages 271–292, 1980.
17. Langer EMV-Technik. Details of Near Field Probe Set RF 2. Website as of April 2013. `http://www.langer-emv.de/en/produkte/prod_rf2.htm`.
18. B. Liu and B. Baas. Parallel AES encryption engines for many-core processor arrays. *Computers, IEEE Transactions on*, 62(3):536–547, 2013.
19. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Workshop on the Security of the Internet of Things-SOCIOT*, 2010.
20. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
21. A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC, 1996.
22. A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *ACM CCS'11*, pages 111–124. ACM, 2011.
23. A. Moradi, M. Kasper, and C. Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *CT-RSA'12*, volume 7178 of *LNCS*, pages 1–18. Springer, 2012.
24. A. Moradi, D. Oswald, C. Paar, and P. Swierczynski. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-engineering. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays – FPGA'13*, pages 91–100, New York, NY, USA, 2013. ACM.
25. NIST. FIPS PUB 197: Advanced encryption standard. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, as of March 24, 2021.
26. D. Oswald and C. Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *Cryptographic Hardware and Embedded Systems – CHES'11*, volume 6917 of *LNCS*, pages 207–222. Springer, 2011.
27. D. Oswald, D. Strobel, F. Schellenberg, T. Kasper, and C. Paar. When Reverse-Engineering Meets Side-Channel Analysis – Digital Lockpicking in Practice. to appear at SAC'13, 2013.
28. Pico Technology. PicoScope 5200 USB PC Oscilloscopes, 2008. `http://www.picotech.com/picoscope5200-specifications.html`.
29. M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES'10*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.
30. S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES'12*, volume 7428 of *LNCS*, pages 23–40. Springer, 2012.
31. D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas. A 167-processor computational platform in 65 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 44(4):1130–1144, 2009.
32. A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz. Energy analysis of public-key cryptography for wireless sensor networks. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 324–328. IEEE, 2005.
33. H. Wang and Q. Li. Efficient implementation of public key cryptosystems on mote sensors (short paper). *Information and Communications Security*, pages 519–528, 2006.
34. E. W. Weisstein. Variance. Mathworld - A Wolfram Web Resource, December 2010. `http://mathworld.wolfram.com/Variance.html`.
35. F. Zhang, R. Dojen, and T. Coffey. Comparative performance and energy consumption analysis of different AES implementations on a wireless sensor network node. *International Journal of Sensor Networks*, 10(4):192–201, 2011.