Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Systems Architecture 50 (2004) 711–727
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

A tamper resistant hardware accelerator for RSA cryptographic applications ☆

G.P. Saggese [a], L. Romano [a,*], N. Mazzocca [b], A. Mazzeo [a]

[a] *Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy*
[b] *Seconda Università degli Studi di Napoli, Via Roma 29, 81031 Aversa (CE), Italy*

Received 10 September 2003; received in revised form 13 January 2004; accepted 21 April 2004
Available online 20 July 2004

Abstract

This paper presents an hardware accelerator which can effectively improve the security and the performance of virtually any RSA cryptographic application. The accelerator integrates two crucial security- and performance-enhancing facilities: an RSA processor and an RSA key-store. An RSA processor is a dedicated hardware block which executes the RSA algorithm. An RSA key-store is a dedicated device for securely storing RSA key-pairs. We chose RSA since it is by far the most widely adopted standard in public key cryptography. We describe the main functional blocks of the hardware accelerator and their interactions, and comment architectural solutions we adopted for maximizing security and performance while minimizing the cost in terms of hardware resources. We then present an FPGA-based implementation of the proposed architecture, which relies on a Commercial Off The Shelf (COTS) programmable hardware board. Finally, we evaluate the system in terms of performance and chip area occupation, and comment the design trade-offs resulting from different levels of parallelism.

Keywords: Field Programmable Gate Arrays (FPGAs); RSA cryptosystems; Security-and performance-critical applications

1. Rationale and contribution

In the recent years, we are witnessing the proliferation of distributed software applications, consisting of autonomous software components, which interact over the network in a fully automated way, towards the provision of crucial functions. In this context, by crucial functions we mean functions whose failure may result in major economic impact and/or loss of human lives. Emerging e-commerce, health-care, and transportation systems are just a few examples of such

☆ This work was partially funded by: the University of Naples Federico II, the Con-sorzio Interuniversitario Nazionale per l'Informatica (CINI), the National Research Council (CNR), the Ministry of University and Research (MIUR) within the framework of projects: SP1 "Sicurezza dei documenti elettronici", and "Middleware for advanced services over large-scale, wired-wireless distributed systems" (WEB-MINDS), and the Regione Campania within the framework "Telemedicina" project. Authors are grateful to Alessandro Cilardo for the fruitful technical discussions.

* Corresponding author. Tel.: +39-081-7683834; fax: +39-081-7683816.
E-mail address: lrom@unina.it (L. Romano).

1383-7621/$ - see front matter
doi:10.1016/j.sysarc.2004.04.002

applications. For these applications to work properly, it is foremost that a whole set of services, collectively called *infrastructure services*, be provided in a reliable and timely fashion. Typical infrastructure services are directory, discovery, and security. Cryptographic applications, which are used to provide *security services*—such as authentication, confidentiality, authorization, non-repudiation, and time-stamping—are thus characterized by challenging requirements, both in terms of security and of performance.

As to performance, it is well known that cryptographic routines consume a huge amount of computing power. Suffices it to say that a 266 MHz Pentium II machine executing the AES (Advanced Encryption Standard) [13] algorithm achieves a throughput of just 26Mbit/s, while consuming 100% of the CPU time [29]. This clearly indicates that there is not enough processing power to handle the 10 gigabit Ethernet standard now coming to market. In this scenario, the availability of a loosely-coupled, highly-specialized hardware co-processor installed in the host machine and dedicated to the execution of security-related tasks, would off-load the main processor, and ultimately boost the performance of the overall system.

As far as security is concerned, the most crucial issue is how to prevent that the private key be disclosed, for this would allow impostures to impersonate a legitimate principal in electronic transactions. The solution adopted in the vast majority of commercial-grade security products is to store the private key in a file on disk, typically in a protected directory, and possibly in encrypted form. This solution has two major drawbacks.

First, it has been proved to be insecure. In [21], the authors demonstrate that a non-highly-skilled attacker—with no equipment but a floppy disk—has good chances to steal the private key during a so-called lunch-time attack. In a typical *lunchtime attack* scenario, the attacker (who can be a secretary, an employee, or a member of the technical staff) sneaks into the manager's office for a few minutes while he or she is away for lunch. This weakness stems from two factors: (i) copies of the key are present—at all times—in many places in the system, such as operating system swap files and temporary files from previous signing sessions (in this cases the key is also likely to be in plain-text form), and (ii) since cryptographic keys are characterized by a much higher entropy (as compared to the rest of the disk data) and since the human eye has quite a good sensibility to entropy, they can be easily identified by simply scanning the disk contents using a general purpose disk utility with a visual output function.

Second, it is characterized by poor performance. Before the key can be used, the file containing it must be retrieved from disk, and possibly decrypted. Retrieval from disk and decryption have a performance cost which exacerbates the inherent performance problems of the application at hand. Such a cost is in general not negligible, and in many cases unacceptable, especially for cryptographic applications which are to operate on-line.

To meet the stringent security and performance requirements of cryptographic applications, it is thus crucial that effective solutions—both from a performance and from a security viewpoint—be provided to: (i) execute cryptographic procedures, and (ii) store and retrieve cryptographic keys. The development of hardware-based devices which provide crypto-processor and/or key-store facilities is thus receiving more and more attention, due to the widely demonstrated high performance and tamper resistance capabilities of hardware.

Examples of state-of-the-art commercial products are: the *nForce* [25] SSL accelerator and the *nShield* [23,26] hardware security module by nCipher, the *SignMaster ISP* [27] by Compaq, the chip sign CS1015 [22], and the *DS5240* and *DS5250* microcontrollers recently announced by Maxim Microcontroller [24] (let alone the many last generation smart cards, which however exhibit quite poor performance).

This paper presents an hardware accelerator which can be used to improve the security and the performance of virtually any RSA cryptographic application. The accelerator integrates two crucial security- and performance-enhancing facilities: an RSA crypto-processor and an RSA key-store. We describe the main functional blocks of the hardware accelerator and their interactions, and comment architectural solutions we adopted for maximizing security and performance while mini-

mizing the cost in terms of hardware resources. We also present an FPGA-based implementation of the proposed architecture, which relies on a Commercial Off The Shelf (COTS) programmable hardware board. Finally, we evaluate the system in terms of performance and chip area occupation, and comment the design trade-offs resulting from different levels of parallelism.

This work makes three important contributions.

The first contribution is the description of the architecture itself. We describe the individual components of the integrated hardware accelerator and their interactions in detail. We provide a thorough treatment of technical challenges we had to face, and a detailed description of the solutions we have implemented. Such solutions are technology independent, and they are thus suited for implementation in virtually any hardware technology. In some cases—such as in the implementation of RSA-specific components—we analyze the subtlest algorithmic issues, and motivate our design choices. This makes a valuable resource for hardware practitioners who have to solve similar problems. That is especially true of a field—such as computer security—where for most commercial products vendors do not provide details about implementation, performance, and development cost. In other cases—such as in the implementation of the circuitry for addressing the trickiest timing issues—we suggest techniques and approaches which suit a variety of hardware design contexts.

The second contribution is the implementation of the architecture using Commercial Off The Shelf (COTS) FPGA technology, namely a Celoxica RC1000 programmable board mounting a Xilinx Virtex-E 2000 FPGA part. We chose COTS FPGA technology for two fundamental reasons. First, evidence is showing that such a technology allows the processing of data at rates in the gigabit range for different encryption algorithms, which is at least one order of magnitude faster than most reported software implementations [28,30]. Second, it is resistant to tampering. Even if an attacker is able to open up the FPGA package and probe (millions and millions of) points without damaging the device, interpreting the bit-stream is—according to major FPGA vendors [4,5]—a "virtually impossible" task, since the irregular row and column pattern of the hierarchical interconnection network exacerbates the inherent complexity of the reverse engineering process. Due to its low-cost and wide availability, COTS FPGA technology is thus the ideal resource for improving the performance and the security of cryptographic applications. This is especially true of applications which are still in their prototype phase—i.e. they are still subject to changes and/or have not yet reached wide-scale deployment—but nevertheless have stringent security and performance requirements. We emphasize that a whole lot of enterprise applications belong to this category.

The third contribution is the analysis of the trade offs—in terms of performance vs. area occupation—which result from different levels of parallelism, i.e. from different values of the digit size of the RSA crypto-processor. Not only we demonstrate that the digit-serial approach we have taken is a powerful technique, but we also provide quantitative data, which can be used by hardware practitioners as a guideline in the purchase of a new FPGA board, and/or in the configuration of an existing board.

We explicitly note that we do not address the issue of how to avoid logical intrusion to the device, i.e. of how to grant access to the device only to authorized applications/users. Avoiding logical intrusion is of course a key issue and we are well aware that the availability of a tamper resistant device would be of no practical use if mechanisms to avoid logical intrusion were not properly enforced. Indeed, we did address the issue of avoiding logical intrusion, but—due to the complexity of the subject and to the lack of space—they are not described in this paper. The interested reader can refer to [7] for a detailed description of the approach we have taken.

The rest of the paper is organized as follows. Section 2 describes the algorithm that we propose for the execution of modular exponentiation, which is the central part of the Rivest–Shamir–Adleman (RSA) algorithm. Symbol names are defined in Appendix A at the end of the paper. Section 3 describes the architecture of the overall system, i.e. the main functional blocks and their

interactions. Section 4 details the architecture of the core part of the device, including the two most crucial system components, namely the RSA crypto-processor and the RSA key-store. Section 5 illustrates how the architecture has been implemented in FPGA technology. Section 6 presents a thorough evaluation of area occupation and performance of the hardware accelerator. Section 7 concludes the paper with final remarks.

2. Proposed algorithm

This section provides a short description of the algorithm that we propose for the execution of modular exponentiation, which is the central part of the Rivest–Shamir–Adleman (RSA) algorithm [14]. A more detailed description can be found in [6].

The modular exponentiation ($X^E \bmod N$) is typically carried out via repeated modular multiplications ($A \cdot B \bmod N$). Our implementation of modular multiplication is an optimized version of the one which was proposed by Montgomery in [17], and improved by Walter in [18].

We assume that N can be represented with K bits, and we take $R = 2^{K+2}$. The N-residue of A with respect to R is defined as the positive integer $\overline{A} = A \cdot R \bmod N$. Montgomery product of residues of A and B, $MonPro(\overline{A}, \overline{B})$, is defined as $(\overline{A}, \cdot \overline{B} \cdot R^{-1}) \bmod N$, that is the N-residue of the desired $A \cdot B \bmod N$. If $A, B < 2N$, the combination of the techniques described in [18] and [15], yields the following radix-2 binary add-shift algorithm for the computation of *MonPro:*

Algorithm 1 (*Radix*-2 *Montgomery Product MonPro*(*A*,*B*)). Given $A = \sum_{i=0}^{K+2} A_i \cdot 2^i$, $B = \sum_{i=0}^{K} B_i \cdot 2^i$, $N = \sum_{i=0}^{K-1} N_i \cdot 2^i$, where $A_i, B_i, N_i \in \{0,1\}$, $A_{K+1}, A_{K+2} = 0$, computes a number in $[0, 2N[$ which is modulo N congruent with desired $(A \cdot B \cdot 2^{-(K+2)}) \bmod N$.

1. $U = 0$
2. `For` $j = 0$ `to` $K + 2$ `do`
3. `if` $(U_0 = 1)$ `then` $U = U + N$
4. $U = (U/2) + A_j \cdot B$
5. `end for`

We emphasize that this algorithm exploits a representation of A and B as a residue class modulo N, thus replacing the division by N operation with a division by R. If R is chosen as a power of 2, such a division is a low-cost operation for binary represented numbers. However, the preprocessing and postprocessing steps, which are needed to convert the numbers to and from the residue based representation, are expensive. Consequently, the cost of the conversion may be prohibitive if the number of modular multiplications is relatively small. Since RSA modular exponentiation entails the execution of a large number of modular multiplications, the intermediate results can be left in residue representation and the conversion is necessary only in the final step. The Montgomery technique is thus very effective in this case.

The exponentiation algorithm for computing $X^E \bmod N$ known as Right-To-Left binary method [16] can be modified to take advantage of the Montgomery product, as described in the following.

Algorithm 2 (*Right-To-Left Modular Exponentiation using Montgomery Product*). Given X, N, and $E = \sum_{i=0}^{H-1} E_i \cdot 2^i$, $E_i \in \{0,1\}$, computes $P = X^E \bmod N$.

1. $P_0 = MonProd(1, R^2 \bmod N)$
2. $Z_0 = MonProd(X, R^2 \bmod N)$
3. `For` $i = 0$ `to` $H - I$ `do`
4. $\quad Z_{i+1} = MonProd(Z_i, Z_i)$
5. $\quad$ `if` $(E_i = 1)$ `then` $P_{i+1} = MonProd(P_i, Z_i)$
6. $\quad\quad$ `else` $P_{i+1} = P_i$.
7. `end for`
8. $P = MonProd(P_H, 1)$
9. `if` $(P \geqslant N)$ `then return` $P - N$
10. $\quad$ `else return` P

The first phase (lines 1–2) calculates residues of initial values 1 and X. The core of the computation is the loop in which modular squares are performed, and the previous partial result P_i is multiplied by Z_i, based on a test performed on the value of the ith bit of E (H is the number of bits comprising E). It is worth noting that, being instructions 4 and 5–6 independent of each other, they can be executed in parallel. Instructions 8–10

allow switching back from the residue domain to normal representation of numbers.

3. Overall system architecture

In this section we describe the architecture of the hardware accelerator, i.e. its main functional blocks and their interactions.

A simplified schematic of the overall system is depicted in Fig. 1. The hardware accelerator is composed of two main parts:

- The *Processing Engine*, which is in charge of executing the cryptographic routines and of storing cryptographic keys. It consists of the RSA key-store, the RSA Processor, and some additional circuitry. The RSA *Processor* is in charge of performing RSA encryption and decryption. The parallelism of the RSA block is S. As S increases, silicon area is traded off for performance. The *key-store* holds the keys which the RSA Processor uses. It has a parallelism of 32 bits. A multiplexer (Mux in Fig. 1) is used to feed the RSA Processor either with data which is to be processed or with the key (over a shared bus). Since there are blocks with a parallelism of 32 bits (the host bus and the key-store) and a block (the RSA Processor) with a possibly higher parallelism, some further logic is needed to allow the blocks to communicate. More precisely, the *Ser2Par* is used to convert 32 bits words to S bits words, whereas the *Par2Ser* converts S bits words to 32 bits words. A detailed description of the two fundamental components of the Processing Engine, i.e. the RSA Processor and the key-store, is given in Section 4.
- The *Interfacing Logic*, which is in charge of handling the communication with the Host-node, i.e. the machine on which the hardware accelerator is mounted. It consists of the Register Interface, the Control and Status registers, the BufferIn and BufferOut components, the Memory Interface and the Scratch-pad Memory. The *Register Interface* allows the Main Controller to exchange data with the *Control* and *Status* registers, which are used by the host node to send commands to the accelerator and by the accelerator to make information about its status available to the host computer. The *Memory Interface* controls the Scratchpad Memory, which is used to exchange data with the host. The keys, the data to be encrypted/decrypted, and the results of the encryption/decryption are transferred from the host memory to the accelerator—and vice versa—through the *Scratch-pad Memory*. The *BufferIn* and *BufferOut* blocks are used to interface the low speed asynchronous Scratch-pad Memory to the circuitry of the Processing Engine, which runs at a higher clock frequency.

An additional component, the *Main Controller*, issues control signals to the Processing Engine and to the Interfacing Logic to coordinate their interactions.

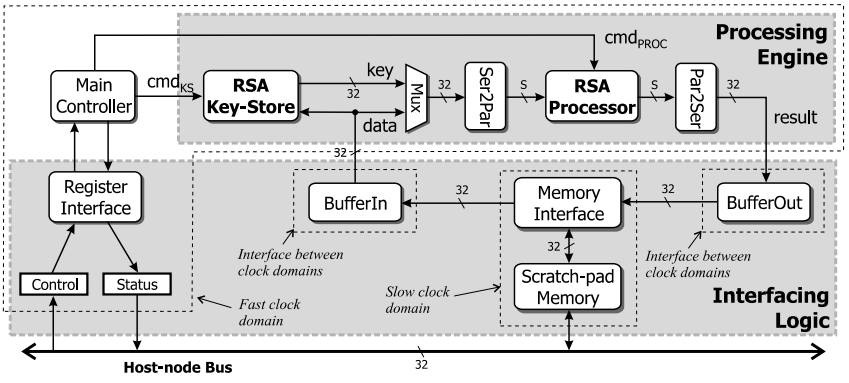

Fig. 1. Overall architecture of the hardware accelerator.

4. Architecture of the processing engine

This section describes the architecture of the Processing Engine, and in particular of its the two crucial components, i.e. the RSA Processor and the RSA key-store.

4.1. Architecture of the RSA processor

From Algorithm 2 of Section 2 it follows the RSA Processor has to properly sequence repeated Montgomery products on data, and to store intermediate results in a register file. Montgomery products are computed according to Algorithm 1 of Section 2, which entails a sequence of micro-operations consisting of loads/stores, shifts, and additions on registers. Operands are long integers whose length is related to K. In our implementation, each instruction is broken into multiple parts and executed in a serial fashion on S-bit operands. Hence, S represents the serialization factor (or in multiprecision arithmetic terminology S is the digit-size). At a high level of abstraction, the RSA Processor thus consists of a *Data-Path Unit*, performing data-processing operations, and a *Control Unit*, which enforces proper sequencing of such operations. The Data-Path is designed to operate on S-bit wide words and it is composed of three macro blocks (Fig. 2): a *Memory Section* block for storing intermediate data, and two processing units—named *P-processor* and *Z-processor*—which run concurrently and implement modular product and modular squaring, respectively. We explicitly note that, since squaring can be performed as a product, the Z-processor is actually a simplified version of the P-processor. That means the P-processor and Memory Section alone are sufficient to implement the algorithm. Hence, the Z-processor is not strictly needed, and it could be eliminated. However this would double the execution time, while scaling down the total area by a factor smaller than 2 (since the area of the Memory Section is constant). We thus chose to include in our architecture both the P and the Z processors, since this solution is characterized by a better value of the product $A \cdot T$ (where A is the total area occupation and T the total execution time). In scenarios where area is a concern and performance is not an issue, the alternative which relies solely on a P-processor can be of interest.

The *Data-Path* is organized as a three-stage pipeline. The first stage fetches the operands. Partial results are read from the Memory Section and from the P-processor and the Z-processor internal registers. The second stage operates on the partial results. The last stage writes results back to registers.

The *Control Unit* is in charge of generating control flow signals (for loops, and conditional and unconditional jumps) for the data-path. It runs concurrently in pipelining with the data-path, in such a way that processing of data and sequencing of operations can be overlapped.

Due to space limitations, in the following we describe in detail only some of the components of the architecture. For a thorough treatment of the components which are not dealt with here, please refer to [6].

The *P-processor* performs several operations: (1) it acts, along with the Memory Section, as a

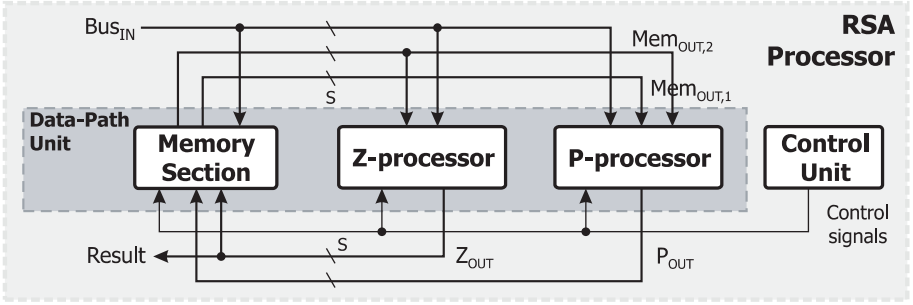

Fig. 2. Overall architecture of the RSA processor.

serial Montgomery multiplier (it implements Algorithm 1); (2) it carries out a preprocessing phase which accelerates the modular product (the computation of $2B + N$); and (3) it performs the reduction step (it ensures that the result is indeed the modulus of the requested exponentiation). The schematic of the P-processor and of the Memory-Section is reported in Fig. 3.

A few comments are in order about the design choices we have made. The digit-serial approach [19,20] has the fundamental advantage of allowing area saving (provided that area-overhead due to serial to parallel data formatting and subsequent inverse conversion does not outweigh the area saving which derives from smaller operands and from a narrower data-path). However this advantage comes at the price of an increase in the overall execution time, as opposed to a fully parallel alternative. Such an increase is due to the fact that the delay of the flip-flops, which are in the critical path, contribute to the overall execution time with a factor 1 in the fully parallel alternative and with a factor S in the serial solution. Evaluating the net effect of these two conflicting factors is not a simple task. For this reason, in Section 6 we conduct a thorough analysis of the design trade-offs in terms of performance vs. area.

As to the P-processor we emphasize that resorting to a preprocessing phase for computing (once for all) the value $2B + N$, results in a shorter execution time, since $2B + N$ is added to $U(K + 3)/4$ times in average (assuming A_i and U_0 are independent and equally distributed). Hence, by adding one register of $K + 3$ bits, we were able to save $M \cdot ((K + 3)/4 - 1)$ clock ticks on the total execution time, and to use a two word adder instead of a three word adder.

As to the *Memory Section*, we explicitly note that each operand is stored as an array of S-bit words. This allows us to avoid the use of multiplexers for selecting the part of the operand which is to be summed. The Memory Section also contains *Register E*, which stores exponent E (up to K bits long).

4.2. Key-store

In this subsection, we briefly describe the internal structure of the key-store component. Further information can be found in [7].

The key-store allows secure storage and fast retrieval of RSA keys for encryption and decryption. The architecture of the key-store depends on the key format of an RSA key-pair. By *key format*, it is meant the set of data items which form the key of a cryptographic algorithm. It includes the private key and the public key, as well as additional data items which may be useful for speeding up the

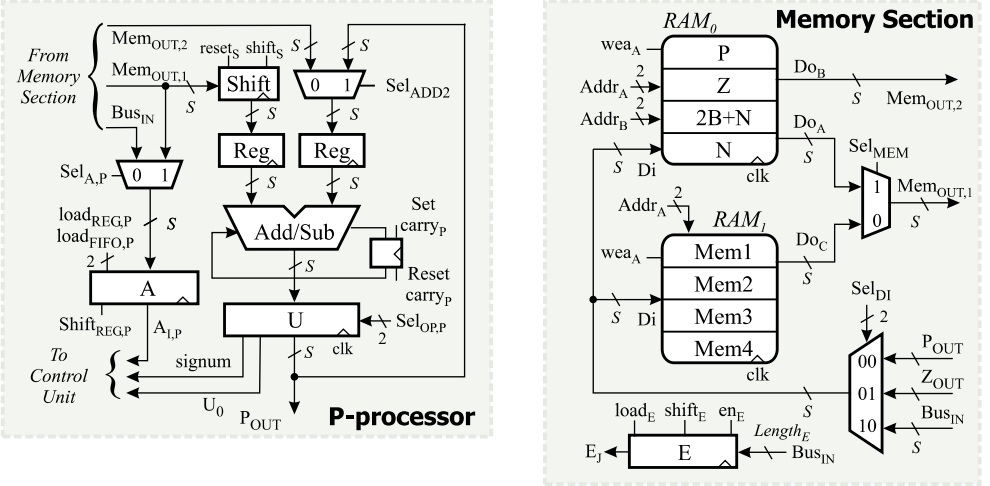

Fig. 3. Structure of the P-processor and of the memory-section.

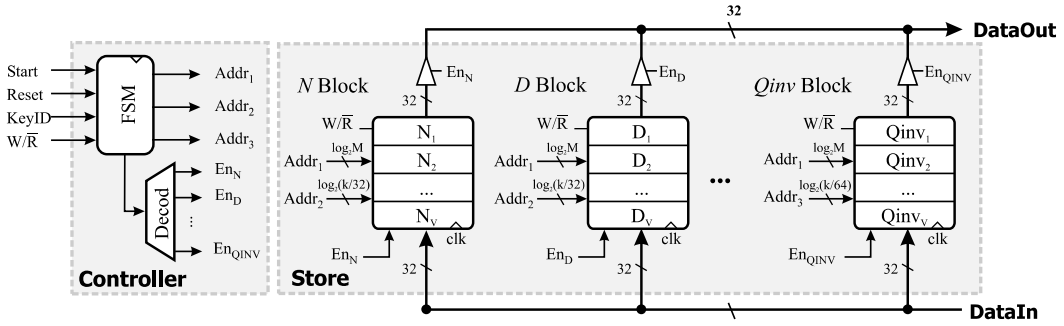

Fig. 4. Organization of the hardware key-store module.

encryption/decryption process. In particular, the latest version of the RSA standard, namely PKCS #1 v2.1 [8], suggests that three pre-computed factors $(\mathrm{d}P = \mathrm{d}\bmod(p-1), \mathrm{d}Q = \mathrm{d}\bmod(q-1)$, and $qInv = q^{-1} \bmod p)$, although not strictly part of the key, be calculated in a pre-processing phase (immediately after the generation of a key pair) and stored with the private key. We followed the suggestions of the PKCS standard and included such precomputed factors in the key format for our key-store. Consequently, in the rest of the paper an RSA key-pair consists of the following items: $p, q, N, E, D, \mathrm{d}P, \mathrm{d}Q$, and $qInv$. It is worth noting that, for the sake of efficiency, E is usually chosen as a short integer, but its length can be up to K bits. The parameters p, q, $\mathrm{d}P$, $\mathrm{d}Q$, and $qInv$ are $K/2$-bit integer numbers. The other components, i.e. N and D, are K-bit long. Given these bit-lengths, it follows that a total of $(11/2) \times K$ bits is required to store an RSA key.

A simplified schematic of the key-store is depicted in Fig. 4. It consists of two parts: the Controller and the Store. Since the data-path parallelism of the RSA Processor is a multiple of 32 bits and the host machine bus is typically 32 bits wide, the key-store has an internal parallelism of 32 bits, too. The *DataOut* bus is used to transfer the keys to the RSA Processor. The key-store can hold up to V RSA keys. The actual value of parameter V must be set before the synthesis step. Individual components of each key (namely $p, q, N, E, D, \mathrm{d}P, \mathrm{d}Q$, and $qInv$) are orderly stored in the memory blocks of Fig. 4, i.e. they are mapped to specific physical memory blocks. More precisely, all N; moduli of the keys—where $i = 1 \ldots V$—are stored vertically in the *N Block*, while individual components of the i-th key are striped over an horizontal row spanning all memory blocks. The *Controller* is composed of a Finite State Machine (FSM) and some decoding logic to enable the individual components of the Store part as needed.

As to key initialization, two options are possible: key values can be hard-wired in the design during the synthesis step, or they can be loaded from the outside once the accelerator has been mounted on the host node (via the DataIn bus).

5. FPGA implementation of the proposed architecture

We implemented the architecture described in Sections 3 and 4 using a Commercial Off The Shelf FPGA board mounting a Xilinx device. This section is organized as follows. In Section 5.1 we describe the board and the FPGA part we used. Then, in Section 5.2, we discuss the implementation of the Processing Engine.

5.1. Target device and design tools

We used a Celoxica RC1000 board. Celoxica RC1000 is a standard full length 32-bit PCI card [2]. The architecture of the board is illustrated in Fig. 5. The board is made up of an FPGA, which has to be configured (programmed) to implement the computational engine, and of a set of memory banks (8 MB of fast asynchronous SRAM, split into four 2 MB memory banks). These can be used

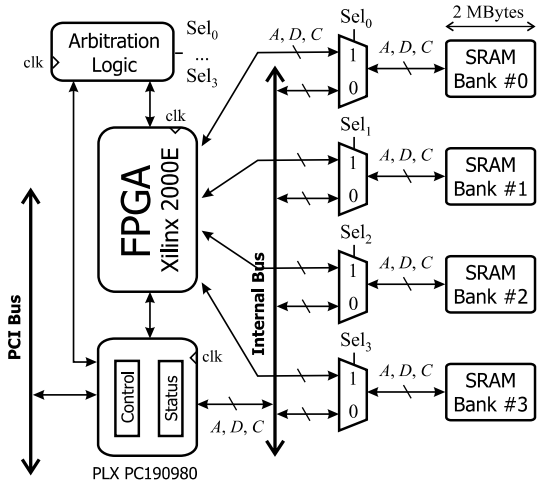

Fig. 5. Celoxica board architecture and connection to the host computer.

as a scratch memory for the FPGA, or as a shared memory area, which is accessible to the FPGA directly and to the host CPU by DMA transfers across the PCI bus.

The Xilinx Virtex series of FPGAs consists of a logic cell—or Configurable Logic Blocks (CLB)—and interconnection circuitry, tiled to form a chip. Each CLB consists of two slices, and each slice contains two 4-inputs Look-Up Tables (LUTs), two flip-flops (FFs), and associated carry chain logic. Each LUT can either be used as a 16x1 bit RAM, or as a 1–16 cycle delay shift register. Each flip-flop has a clock enable and a reset signal. Both the LUT and the flip-flop can be accessed independently. The FPGA we used is a Xilinx VirtexE2000-8 [3]. It has 19200 slices, 19520 tristate buffers, and incorporates 160 fully synchronous dual-ported 4096 bit block memories named Block SelectRAM (BRAM).

As far as design tools are concerned, we used VHDL (specifically Aldec Active VHDL 4.2) for the design of the most critical parts of the FPGA, since it allows finer control of the synthesized hardware, thus making architecture-specific optimizations possible. We resorted to Handel-C [1]—an high-level C-like language—for the blocks whose functionality is best expressed at a higher level of abstraction, such as the Main Controller, the Register Interface, and the Memory Interface. We used Synplicity Synplify Pro 7.1 for synthesis, integrated in the Xilinx ISE 4.1 design flow.

As to the design process, the preliminary floorplan of the blocks was carried out manually, since the design environment does not provide features for automatically handle this issue.

5.2. Implementation of the processing engine

In this section we detail how we implemented the conceptual architecture of the Processing Engine described in Section 4, on top of the FPGA device described in Section 5.1.

We do not illustrate all components in detail due to lack of space, but the interested reader can refer to [6], and [7] for a thorough treatment of technical issues.

As to the *RSA processor*, the implementation of some of the blocks of Fig. 3 was relatively straightforward. As an example, the Add/Sub blocks were implemented using dedicated carry chains, and the RAM blocks were mapped to LUTs configured as distributed memories. For some other blocks, such as registers A and U of both processors, and register E, we had to find efficient implementation schemes in order to reduce area occupation. We emphasize that we exploited, whenever possible, architecture-specific features of the Xilinx programmable device to maximize the performance while minimizing area occupation. As an example, the Virtex architecture has plenty of tristates which can be used as routing resources and a relatively limited amount of LUTs, which can be considered as a general purpose resource. When implementing multiplexers, it would be desirable to use tristates. However, tristate-based multiplexers show quite larger delays than their LUT-based counterparts. We thus chose tristate-based solutions only in cases where it was possible to slow down the cycle period of stages without reducing the operation frequency of the overall pipeline.

As to the *key-store*, we exploited the availability in the Virtex architecture of dedicated memory blocks (Block SelectRAM). This resulted in reduced area occupation for two main reasons: (1) it requires about $\lceil((11/2)\cdot K\cdot V)/4096\rceil$ BRAMs (since each BRAM can store 4096 bits) to store V

RSA keys on K bits, whereas the implementation based solely on flip-flops requires $(11/4) \cdot K$ slices; and (2) with this approach, the internal logic of memories can be exploited to avoid the use of multiplexers for selecting the sub-words of a key item.

The Ser2Par, Mux, and Par2Ser adjust the different parallelism of the components, thus making the interconnection of the individual blocks of the Processing engine possible. The *Ser2Par* block is composed of $S/32$ registers, each one consisting of 32 bits. They are loaded one at a time, selecting with the enable signals which portion of S bit register to load, and then the S bit result is showed in parallel to the RSA Processor. The *Par2Ser* block is made up of $S/32$ 32-bit registers, and $S/32$ 2-to-1 multiplexers (32-bit wide), the multiplexers allow parallel loading of registers and configuration of the structure as a 32-bit shifting queue. The Ser2Par and Par2Ser blocks are also used as registers to break up the propagation of signals between the connected blocks, thus allowing the Processing Engine to work at a high clock frequency.

5.3. Implementation of the interfacing logic

In this section, we detail how we implemented the conceptual architecture of the interfacing logic described in Section 3 on top of the FPGA device described in Section 5.1.

The Register Interface, Memory Interface, Buffer-In, and Buffer-Out blocks connect the Processing Engine to the Control and Status register, and to the Scratch-pad Memory, which are built-in resources of the RC1000 board.

The Scratch-pad Memory is used as a shared buffer to move data (the keys, the raw data to be encrypted or decrypted, and the results of the processing) to/from the Processing Engine, since the board does not allow the host to gain direct access to the FPGA. To move the data, a DMA is activated from the central memory to the Scratch-pad Memory, i.e. the RC1000 board RAM (and vice versa). Then, the Main Controller block on the FPGA uses the Memory Interface, which is an interface for asynchronous RAMs, to move the data from the Scratch-pad Memory to the Processing Engine, which is on the FPGA.

As to the *Scratch-pad Memory*, it is worth noting that the RC1000 RAM memories are asynchronous and the maximum sustainable operation frequency, when controlled by the FPGA, is about 21 MHz, while the Processing Engine can run at a higher frequency (up to 110 MHz). Consequently, buffers are needed to cope with the frequency gap, in order to avoid that the slower sub-system slow down the overall circuit. In other words, the system is composed of two distinct clock domains, namely the Fast clock domain and the Slow clock domain, as emphasized in Fig. 1. The solution we adopted to interface these two clock domains is to use two single-port memories (*BufferIn* and *BufferOut*) which can be controlled both by the Memory Interface (in the slow clock domain), and by the Processing Engine (in the fast clock domain). Fig. 6 illustrates the structure of a generic RAM block of $2^R \times 32$ bit words, which can act as a BufferIn, being accessed in write mode (when $Owner = 0$) from the slow clock (Clk_{Slow}) domain and in read mode (when $Owner = 1$) from the fast clock (Clk_{Fast}) blocks. The structure of the BufferOut is similar (and it is thus omitted here).

The *Main Controller* is in charge of arbitrating the ownership of the buffers using the $Owner_{SET}$ and $Owner_{LOAD}$ signals. Both the BufferIn and the BufferOut are implemented using concentrated RAMs (Block SelectRAMs). The Main Controller uses the Memory Interface to control the Scratch-pad Memory in order to move the data from the

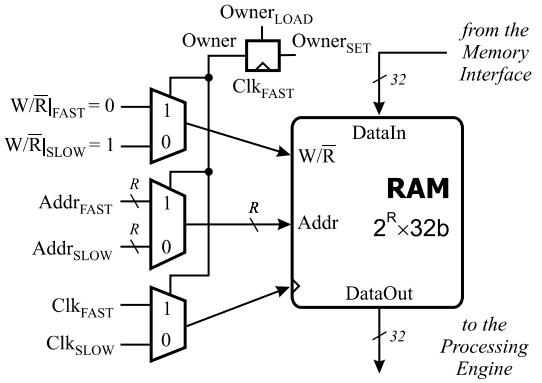

Fig. 6. Structural view of a general BufferIn facilities between two clock domains.

RC1000 on-board RAMs to the BufferIn and from the BufferOut to the on-board RAMs. Since the Main Controller and Memory Interface belong to different clock domains, they communicate using handshake signals. The Main Controller handles the ownership of the BufferIn and BufferOut, which are granted to the Memory Interface or to the Processing Engine during individual phases of the data transfers.

The *Register Interface* is used by the Main Controller to show the status of the hardware accelerator and to receive commands from the host.

6. System evaluation

In this section we conduct a thorough analysis of the implementation trade-offs—in terms of performance vs. area—of the proposed hardware accelerator architecture, as a function of the serialization factor S. The results of this analysis can be used as a guideline by FPGA designers to choose the compromise solution which best satisfies the constraints of specific contexts.

The effectiveness of the alternative solutions was evaluated against two conflicting sets of metrics:

(1) *time* metrics—the time T_{Enc} needed by the hardware accelerator to perform an RSA encryption, and the time T_{Dec} needed to perform an RSA decryption;
(2) *area* metrics—the number of basic building blocks of the Xilinx Virtex-E architecture which are needed to implement the system, namely the number of slices N_{Slices}, and the number of Select BlockRAMs N_{BRAMs} (these metrics are the equivalent of the gate count, or the silicon area, in a VLSI circuit).

The section consists of three sub-sections. In Section 6.1 we evaluate the performance of the system. In Section 6.2 we evaluate area occupation. Finally, Section 6.3 comments the performance vs. area trade-offs of the FPGA implementation, as a function of the serialization factor S.

6.1. Performance evaluation

In this section we report experimental data regarding the time needed for a 1024-bit RSA encryption and RSA decryption, i.e. T_{enc} and T_{Dec} respectively, as a function of S. The time T_{Enc} is given by the formula

$$T_{Enc} = N_{KS} \cdot T_F + N_{Data} \cdot (T_F + T_s) + N_{Enc} \cdot T_F + N_{Res} \cdot (T_F + T_s) \quad (1)$$

In formula (1), the meaning of symbols is as follows: N_{KS} is the number of clock ticks in the fast clock domain needed to move the key from the key-store to the RSA Processor; N_{Data} is the number of clock ticks needed to move the data to be encrypted from the Scratch-pad Memory to the RSA Processor; N_{Enc} is the number of clock ticks for an RSA encryption; N_{Res} is the number of clock ticks to move the result from the RSA Processor of the modular exponentiation to the Scratch-pad Memory; finally T_F and T_S are the periods of the clocks of the fast clock domain and of the slow clock domain, respectively.

The time T_{Dec} is given by the formula

$$T_{Dec} = N_{KS} \cdot T_F + N_{Data} \cdot (T_F + T_s) + N_{Dec} \cdot T_F + N_{Res} \cdot (T_F + T_s) \quad (2)$$

Please note that formula (2) can be derived from formula (1), provided that the number of fast clock ticks needed for an encryption (N_{Enc}) be replaced with the number of clock ticks for a decryption (N_{Dec}).

We emphasize that some data transfers consist in a two phase data movement through the BufferIn and BufferOut components, as described in Section 5.3. That means such data transfers involve movements of data across the boundaries of the two clock domains of the system, i.e. the fast clock domain and the slow clock domain. This explains the term $T_F + T_S$ which appears in formulas (1) and (2).

In order to evaluate T_{Enc} and T_{Dec}, we proceed as follows. First, we briefly analyze physical actions which are related to the individual contributions which appear in formulas (1) and (2). This allows us to compute the number of clock ticks for each operation, i.e. $N_{KS}, N_{Enc}, N_{Dec}, N_{Data}$, and N_{Res}.

Second, we conduct a time analysis of the system. This allows us to estimate the values of the clock periods, i.e. T_F and T_S.

N_{KS} is the number of clock ticks for moving the key items from the key-store to the RSA Processor. Since the total amount of data involved is 2×1024 bits (to move the modulus N and E or D, respectively for encryption and decryption), which are transferred 32 bits at a time, N_{KS} equals 64, independently of S.N_{Data} and N_{Res} have the same value, since they are related to the movement of the same amount of data (1024 bits) over equivalent data-paths (i.e. data-paths operating at the same frequency and with the same parallelism). Since the parallelism of the data-paths is 32 bits, they are both equal to 32, independently of S.

As to N_{Enc}, it is possible to compute the number of clock cycles of an RSA encryption, by analyzing the operation of the architecture of the RSA Processor. The value N_{Enc} is given by the following formula:

$$N_{Enc} = ((2M + 2) + (K + 3) \cdot M) \cdot (len(E) + 2) + 2M + 1 \quad (3)$$

where M is equal to $\lceil (K + 3)/S \rceil$.

The formula has been derived taking into account that:

- an encryption entails the execution of a modular exponentiation, which in turn entails the execution of $len(E) + 2$ modular products (where E is the exponent), plus a few minor operations, namely the reduction step (M clock ticks) and the transfer of the final result ($M + 1$ clock ticks);
- the execution of a single modular product, entails two phases. First, A must be loaded to Register A, and the preprocessing phase for the computation of $2B + N$ must be executed. This phase takes $(2M + 2)$ clock ticks. Second, the j loop of Algorithm 1 must be executed. This takes $(K + 3) \cdot M$ clock ticks, since the body of the loop consists of an addition which takes M clock ticks, and it is executed $K + 3$ times.

As to formula (2), we just need to compute N_{Dec}, which can be derived from formula (3) by replacing $len(E)$ with $len(D)$, where $len(D)$ is the length of the private exponent.

We followed the suggestions of the PKCS standard to use a short prime integer as the exponent E (more precisely we chose $E = 7$). As to the length of the private exponent, $len(D)$, this is equal to the number of bits of the modulus N, i.e. 1024.

The values of N_{Enc} and N_{Dec}, obtained by plugging these values into formulas (2) and (3), are reported in Table 1, for different values of the serialization factor S.

The first phase of the analysis, i.e. the computation of $N_{KS}, N_{Enc}, N_{Dec}, N_{Data}$, and N_{Res}, is thus concluded. We now have to proceed with the second phase, i.e. the estimation of the values of the clock periods, T_F and T_S. This entails conducting a thorough timing analysis of the system, since we have to identify the limiting factor, i.e. the slowest block of each domain, which determines the actual operation frequency for the domain.

The following relationships must be satisfied:

$$T_F \geqslant \max\{T_{CK,X}\} \quad (4)$$

$$T_S \geqslant \max\{T_{CK,Y}\} \quad (5)$$

$$T_S = T_F \times Q \quad (6)$$

where $T_{CK,X}$ is the minimum clock which block X can sustain. Relationship (4) must be evaluated for every block X belonging to the fast clock domain and for every block X belonging to the interface domain (i.e. BufferIn and BufferOut). Relationship (5) must be evaluated for every block Y belonging to the slow clock domain and to the interface domain. Finally, the rationale behind relationship (6) is that the best practice, if multiple clock signals are needed, is to derive any additional clock from the fastest clock available in the circuit.

Table 1
The number of clock ticks N_{Enc} and N_{Dec} for 1024-bit RSA encryption/decryption operations (with $E = 7$)

S	N_{Enc}	N_{Dec}
256	25,746	5,280,833
128	46,334	9,503,857
64	87,510	17,949,905
32	169,862	34,842,001

Table 2
Minimum clock periods T_{CK} (in ns) for the blocks composing the fast clock domain and the interface domain

S	RSA Proc	Par2Ser	Ser2Par	Key-Store	Mux	Register Interface	Main Controller	BufferIn	BufferOut
256	30.9	6.35	7.19	7.31	4.67	7.70	6.03	5.62	5.62
128	18.6	5.99	7.40	7.31	4.67	7.70	6.03	5.62	5.62
64	11.6	5.06	6.46	7.31	4.67	7.70	6.03	5.62	5.62
32	8.74	5.02	6.14	7.31	4.67	7.70	6.03	5.62	5.62

As to relationship (4), the value of T_F must be chosen as the largest value in Table 2. Please note that some values are independent of S.

It is worth noting that the limiting factor for the maximum clock rate of the Processing Engine and of the Interfacing Logic, is the clock cycle which the RSA Processor can sustain, independently of the value of the factor S. For this reason, we optimized the RSA Processor for each S, and then we used its maximum clock frequency as the target clock frequency for the other blocks. This allowed the synthesis tool to slow down the blocks which are not in the critical path, thus favoring area saving. We used a data flow aware placement of the blocks on the FPGA and exploited flip-flops and registers in order to split the propagation delays around the RSA Processor. The final result is that T_F, the frequency of operation of the fast clock for the overall system (see Table 4) is very close to the frequency of operation of the RSA Processor alone.

As to relationship (5), the value of T_S must be chosen as the largest value in Table 3.

Not surprisingly, the frequency of operation of the slow clock domain is limited by the maximum frequency of accesses sustained by the On-board RAM. We measured the maximum frequency of data transfers between the On-Board RAM and the internal buffers. We found that it is about 21 MHz. This value is the result of several contributions: the maximum sustainable access rate of the On-Board RAM, the delay of the block Memory Interface, and the propagation delay due to the parasitic capacitances of the wires among the FPGA part and the pins of the RAMs.

To derive the slow clock from the main clock of the board—which is used as the fast clock—we used the Delay Locked Loop (DLL) component, available in Xilinx FPGAs, to divide the frequency of the main clock. The advantage of using the Xilinx DLL is that it exhibits a very small skew between the clocks and has a dedicated output circuitry to control low-capacitance nets dedicated to clock distribution. The disadvantage is that not all values are possible for the dividing factor Q. More precisely, Q must belong to the set $\{1.5; 2; 2.5; 3; 4; 8\}$.

From the analysis of values in Tables 2 and 3, and taking into account what are the possible values for Q, we solved the equation system consisting of relationships (4)–(6). Results are reported in Table 4, as a function of S.

All terms which appear in formulas (1) and (2) are now known, and they can be plugged in the formulas. The values of the performance metrics T_{Enc} and T_{Dec} are reported in Table 5 as a function of the serialization factor S.

Not surprisingly, the performance of the system improves as the serialization factor S increases.

Table 3
Minimum clock periods T_{CK} (in ns) for the blocks composing the slow clock domain and the interface domain

	Memory Interface	BufferIn	BufferOut	Scratch-pad Memory
T_{CK}	7.43	5.62	5.62	47.6

Table 4
Clock periods T_F and T_S, and dividing factor Q, as a function of the serialization factor S

S	T_F [ns]	T_S [ns]	Q
256	32.2	48.3	1.5
128	19.8	49.5	2.5
64	13.0	52	4
32	9.8	78.4	8

Table 5
Performance metrics T_{Enc} and T_{Dec} for 1024 bit encryption and decryption, as a function of the serialization factor S

S	T_{Enc} [ms]	T_{Dec} [ms]
256	0.837	170.0
128	0.924	188.1
64	1,143	233.3
32	1.671	341.4

6.2. Area evaluation

In this section we provide data about the area occupation of the overall system as a function of S.

Table 6 reports the values of the area metrics N_{Slices} and N_{BRAMs} as a function of S for individual system components. Please note that the amount of resources due to parallel and serial conversions is much less (between 8% and 10%), as compared to the amount of resources needed for implementing the computing part.

The area occupation of the overall system is reported in Table 7, as a function of the serialization factor S. Comparing Tables 6 and 7, it is clear that the RSA Processor alone contributes for the most part to the area occupation of the overall system. We explicitly note that one of the area metrics, namely N_{BRAMs}, is independent of S. We emphasize that the proposed implementation uses a percentage ranging about from 6% to 20% of the total VirtexE-2000 resources.

6.3. Trade-off analysis

In this section we comment performance vs. area trade-offs, as a function of the serialization factor S. The results of this analysis can be used as a guideline by FPGA designers to choose the compromise solution which best satisfies the constraints of specific contexts.

Table 7
Area occupation of the overall FPGA system in terms of slices and BRAMs, as a function of the serialization factor S

S	N_{Slices}/percentage of total	N_{BRAMs}/percentage of total
256	3499/18.2	8/5
128	2315/12.1	8/5
64	1550/8.1	8/5
32	1327/6.9	8/5

Since one of the area metrics, namely N_{BRAMs}, is independent of S, we only analyze the trade-offs between the two performance metrics T_{Enc} and T_{Dec}, and N_{Slices}. The trade-offs between area occupation and time performance of the overall FPGA system are illustrated in Fig. 7, which reports the different design points as a function of the serialization factor S.

As S increases, both T_{Enc} and T_{Dec} decrease (i.e. the performance of the system improves), while N_{Slices} increases (i.e. the area occupation of the system also increases).

We explicitly note that the solution with the smallest parallelism ($S = 32, T_{Enc} = 1.671$ ms and $T_{Dec} = 341.5$ ms), requires just 6.9% of the total resources available on a VirtexE-2000 device. That means this solution is suitable for applications with stringent budget constraints, since it could be implemented even on a much cheaper device, such as a Xilinx VirtexE-200 or a SpartanII-150E. At the opposite end ($S = 256, T_{Enc} = 0.837$ ms and $T_{Dec} = 170.1$ ms), the area consumption is still less than 20% of the total resources available on a Virtex 2000E device. That means that this solution is suitable for applications with challenging performance requirements. For these applications, the FPGA designer can choose $S = 256$, and then tile multiple instances of the Processing Engine on a single FPGA board, thus achieving an a aston-

Table 6
Area occupation for individual blocks of the accelerator. In each column, values are measured in Xilinx slices (left) and BRAMs (right)

S	RSA Proc	Par2Ser	Ser2Par	Key-Store	Mux	Memory Interface	Register Interface	BufferIn	BufferOut	Main Controller
256	2902/0	156/0	147/0	22/6	48/0	117/0	32/0	16/1	16/1	43/0
128	1870/0	82/0	69/0	22/6	48/0	117/0	32/0	16/1	16/1	43/0
64	1188/0	38/0	30/0	22/6	48/0	117/0	32/0	16/1	16/1	43/0
32	995/0	19/0	19/0	22/6	48/0	117/0	32/0	16/1	16/1	43/0

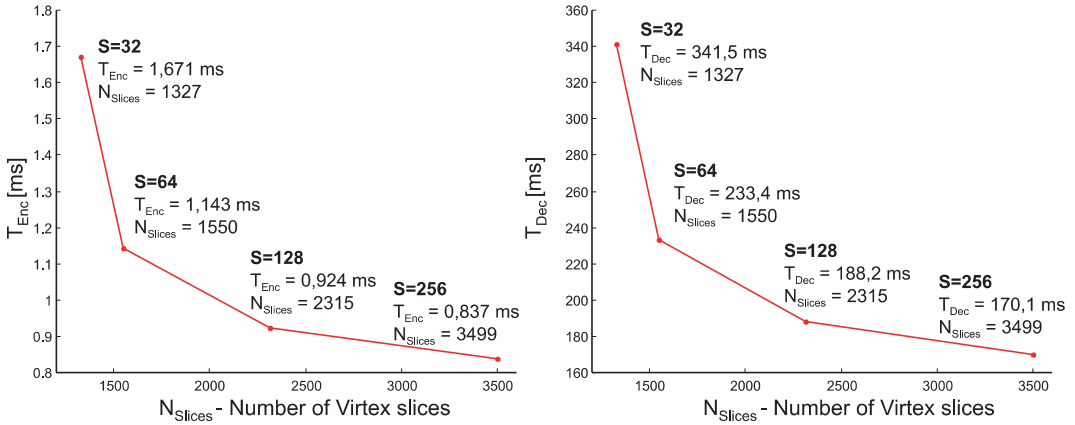

Fig. 7. Area occupation and time performance of the overall FPGA system as a function of the serialization factor S.

ishingly high processing bandwidth (up to 6.2 Mbit/s encrypting, and 30.1 Kbit/s in decrypting).

7. Conclusions

In this paper we addressed a key issue: how to meet the challenging performance and security requirements of cryptographic applications.

The paper advocates an hardware-based approach which relies on an integrated architecture for a custom hardware device, which implements high performance and tamper-resistant RSA processor with associated key-store. Since RSA is the most widely adopted standard for cryptographic systems, our accelerator can effectively be used to improve the dependability (namely security and performance) of a wide class of security services.

This work makes three important contributions. The first contribution is the description of the architecture itself. We described the individual components of the integrated hardware accelerator and their interactions in detail. We provided a thorough treatment of technical challenges we had to face, and a detailed description of the solutions we have implemented. Such solutions are technology independent, and they are thus suited for implementation in virtually any hardware technology. In some cases—such as in the implementation of RSA-specific components—we analyze the subtlest algorithmic issues, and motivate our design choices. This makes a valuable resource for hardware practitioners who have to solve similar problems. That is especially true of a field—such as computer security—where for most commercial products vendors do not provide details about implementation, performance, and development cost. In other cases—such as in the implementation of the circuitry for addressing the trickiest timing issues—we suggest techniques and approaches which can be exploited in a variety of hardware design contexts.

The second contribution is the implementation of the architecture using Commercial Off The Shelf (COTS) FPGA technology, namely a Celoxica RC1000 programmable board mounting a Xilinx Virtex-E 2000 FPGA part. We chose COTS FPGA technology for two fundamental reasons. First, evidence is showing that such technology allows the processing of data at rates which are at least one order of magnitude larger than most reported software implementations. Second, it is resistant to tampering. Due to its low-cost and wide availability, COTS FPGA technology is thus the ideal resource for improving the performance and the security of cryptographic applications. This is especially true of applications which are still in their prototype phase—i.e. they are still subject to changes and/or have not yet reached wide-scale deployment—but nevertheless have stringent security and performance requirements. It is worth emphasizing that a whole lot of enterprise applications belong to this category.

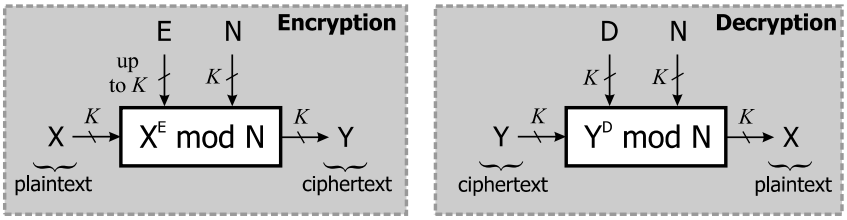

Fig. 8. RSA encryption and decryption.

The third contribution is the experimental analysis of the trade offs—in terms of performance vs. area occupation—which result from different levels of parallelism, i.e. from different values of the digit size of the RSA crypto-processor. Not only we demonstrated that the digit-serial approach we have taken is a powerful technique, but we also provided quantitative data, which can be used by hardware practitioners as a guideline in the purchase of a new FPGA board, and/or in the configuration of an existing board.

Appendix A. Overview of the RSA algorithm

The RSA algorithm is a public key cryptographic scheme which is used to implement a variety of security functions. It forms the basis of a number of drafts, recommendations, and de-facto standards for digital signature (PKCS #1 [8], ANSI X9.31 [11], IEEE P1363 [9] and the addendum P1363a [10], and NIST FIPS 186-2 [12]), and it is used by the most widely adopted protocols for secure Internet connection—i.e. SSL [31] and its newer version TLS [32]—in the handshake phase. Fig. 8 illustrates the two possible operation modes of an RSA-based cryptosystems, namely encryption and decryption.

Encryption is the process of transforming plain-text X into cipher-text Y. To encrypt a message X, the RSA algorithm [14] requires the computation of $Y = X^E \mathrm{mod} N$. Decryption consists in transforming cipher-text Y to plain-text X. It entails the calculation of $X = Y^D \mathrm{mod} N$. The algorithm relies on the existence of a key-pair, i.e. of a private key and of a public counterpart associated to it. The fundamental property of a key pair is that given the public key it is computationally in-feasible to derive its private key counterpart. The strength of an RSA cryptosystem is bound to the length of the parameter K, i.e. the length in bits of the modulus N. A modulus of at least 768 bits is recommended, but one had better use for long-term security, 1024-bit or larger moduli (up to 4096). The private key consists of two large primes p and q and an exponent D, such that $D = E^{-1} \mathrm{mod}(p-1) \cdot (q-1)$, where E is the exponent of the public key. The public key consists of the modulus N (which is the product of the p and q private key components) and of the exponent E.

References

[1] Celoxica Ltd web site, datasheets available from <http://www.celoxica.com/tech/handel-c/default.asp>.
[2] Celoxica Ltd web site, datasheets from <http://www.celoxica.com/techmcal_library/files/CEL-MRKDATRC1000-RC1000>.
[3] Xilinxweb site, datasheets from <http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp>.
[4] Xilinx, Configuration Issues: Power-up, Volatility, Security, Battery Back-up, in: Application Note XAPP 092, 1997.
[5] QuickLogic, QuickNote # 57: High-Level Design Security with QuickLogic Devices, 1997.
[6] A. Mazzeo, N. Mazzocca, L. Romano, G.P. Saggese, FPGA-based Implementation of a Serial RSA Processor, in: Proceedings of the Design And Test Europe (DATE) Conference 2003, pp. 582–587.
[7] G.P. Saggese, L. Romano, N. Mazzocca, A. Mazzeo, An FPGA-based key-store for improving the dependability of security services, to appear in Journal of Web Engineering, Rinton Press, Princeton, New Jersey (USA).
[8] RSA Laboratories, PKCS #1 v2.1: RSA Cryptography, Standard Draft 2, January 2001.
[9] IEEE Std 1363–2000: Standard specifications for Public-Key cryptography. IEEE, August 2000.
[10] IEEE P1363 Working Group. IEEE P1363a D6: standard specifications for Public-Key cryptography: additional techniques. November 2000. Available from <http://grouper.ieee.org/groups/1363/P1363a/draft.html>.

[11] ANSI X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA), September 8, 1998.
[12] National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS) Publication 186-2, Digital signature standard, NIST/US Department of Commerce, January 2000.
[13] National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS) Publication 197, Advanced Encryption Standard (AES), NIST/US Department of Commerce, November 2001.
[14] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining Digital Signature and Public-Key cryptosystems, Commun. ACM 21 (1978) 120–126.
[15] Ç.K. Koç, High-speed RSA Implementation, Technical Report TR 201, RSA Laboratories, November 1994.
[16] D.E. Knuth, in: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, MA, 1981.
[17] P.L. Montgomery, Modular multiplication without trial division, Math. Comput. 44 (170) (1985) 519–521.
[18] C.D. Walter, Systolic modular multiplication, IEEE Trans. Comput. 42 (3) (1993) 376–378.
[19] R. Hartley, P. Corbett, Digit-serial processing techniques, IEEE Trans. Circuits Syst. 37 (6) (1990) 707–719.
[20] K.K. Parhi, A systematic approach for design of digit-serial signal processing architectures, IEEE Trans. Circuits Syst. 38 (4) (1991) 358–375.
[21] A. Shamir, N. van Someren, Playing hide and seek with stored key, Financial Cryptography, 1999.
[22] Chip Sign CS1015/Rubicon Chip datasheet. Available from <http://www.chipsign.com/cs1015_rubicon.htm>.
[23] nCipher web site. Available from <http://www.ncipher.com/nfast/index.html>.
[24] Maxim Integrated Products, datasheets available from <http://www.maxim-ic.com/quick_view2.cfm/qv_pk/3339>.
[25] nCipher nForce, datasheets available from <http://www.ncipher.com/nforce/>.
[26] nCipher nShield, datasheets available from <http://www.ncipher.com/nshield/>.
[27] Compaq Atalla SignMaster, datasheets available from <http://atalla.nonstop.compaq.com/object/asgnmspd.html>.
[28] P. Bellows, J. Flidr, T. Lehman, B. Schott, K.D. Underwood, GRIP: a reconfigurable architecture for host-based gigabit-rate packet processing, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 121–130.
[29] M. Roe, Performance of protocols, in: Proceedings of the 7th International Workshop Security Protocols 2000, Lecture Notes in Computer Science, pp. 140–152.
[30] A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists, IEEE Trans. VLSI Syst. 9 (4) (2001) 545–557.
[31] Netscape SSL Specification 3.0, November 1996. Available from <http://wp.netscape.com/eng/ssl3/index.html>.
[32] The TLS Protocol Version 1.1, http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-02.txt, October 2002.

Giacinto Paolo Saggese is currently a post-doc at the University of Illinois at Urbana-Champaign, U.S.A. His research interests include the design, the implementation, and the experimental evaluation of dependable, high-performance systems implemented as FPGAs and ASICs. He graduated in Electrical Engineering in 2000 at the University of Naples Federico II, Italy. He got his PhD in Electrical and Computer Engineering from the same university, in 2004. During his PhD program he has been doing research with Prof. A. Mazzeo, Prof. L. Romano, and Prof. R.K. Iyer.

Luigi Romano is currently an Assistant Professor at the University of Naples. His research interests include some of the fundamental aspects of computer system dependability, namely availability, reliability, performability, and security. He has been at the Center for Reliable and High-Performance Computing of the University of Illinois at Urbana Champaign for eighteen months, doing research with Prof. R.K. Iyer. He has also worked as a consultant for Ansaldo Trasporti and Ansaldo Segnalamento Ferroviario in the field of safety critical computer systems design and evaluation. He received his MS degree in Electronic Engineering and Ph.D. degree in Computer Science from the University of Naples.

Nicola Mazzocca is currently a Full Professor at the Department of Information Engineering of the Second University of Naples. His research interests include dependable and parallel computing, performance evaluation, and embedded systems. He graduated in Electronic Engineering at the University of Naples "Federico II" in 1987. He spent one year at Ansaldo Trasporti, where he worked on the design of process control systems. In 1988 he joined the Computer Science Department of the University of Naples, where he received his Ph.D. degree in Computer Science.

Antonino Mazzeo is a Full professor at the Computer Science Department of the University of Naples Federico II, where he teaches Computer Architectures. His research activity has focused on the following main topics: performance evaluation of computing systems; computer networks and communication protocols; concurrent languages and parallel programming environments; general and special purpose parallel architectures and applications; secure and fault-tolerant computing. Prof. Mazzeo has been a consultant for and has lead major research programs in conjunction with technology leading industries in Italy and abroad.