XTR Implementation on Reconfigurable Hardware

Eric Peeters[1], Michael Neve[1⋆] and Mathieu Ciet[2]

[1] UCL Crypto Group
Place du Levant, 3
1348 Louvain-La-Neuve, Belgium.
{peeters, mneve}@dice.ucl.ac.be – http://www.dice.ucl.ac.be/crypto
[2] Innova Card,
Avenue Coriandre, 13 600 La Ciotat, France.
mathieu.ciet@innova-card.com

Abstract. Recently, Lenstra and Verheul proposed an efficient cryptosystem called XTR. This system represents elements of $\mathbb{F}_{p^6}^*$ with order dividing p^2-p+1 by their trace over $\mathbb{F}_{p^2}$. Compared with the usual representation, this one achieves a ratio of three between security size and manipulated data. Consequently very promising performance compared with RSA and ECC are expected.
In this paper, we are dealing with hardware implementation of XTR, and more precisely with Field Programmable Gate Array (FPGA). The intrinsic parallelism of such a device is combined with efficient modular multiplication algorithms to obtain effective implementation(s) of XTR with respect to time and area.
We also compare our implementations with hardware implementations of RSA and ECC. This shows that XTR achieves a very high level of speed with small area requirements: an XTR exponentiation is carried out in less than 0.21 ms at a frequency beyond 150 MHz.

Keywords: Public key cryptosystem, XTR, reconfigurable hardware, efficient implementation.

1 Introduction and Basics

Nowadays more and more applications need security components. However, these requirements should not interfere with the performance, otherwise security would be disregarded. Ideally, the best solution is when security does not penalize the application. However, two ways are possible to achieve this characteristic: design efficient primitive algorithms and/or try to find fast and optimized implementations of existing algorithms.

XTR, first presented in [12], has been designed as a classical discrete logarithm (crypto)system, see also [11]. However, element representation is done in a special form that allows efficient computation and small communications. This

⋆ Supported by the FRIA Belgium fund.

system also has the advantage of very efficient parameter generations. As shown in [26], the performance of XTR is competitive with RSA in software implementations, see also [7] for a performance comparison of XTR and an alternative compression method proposed in [22]. Mainly two kinds of implementation have to be distinguished: software and hardware. The latter generally allows a very high level of performance since "dedicated" circuits are developed. Moreover it also provides designers with a large array of implementation strategies. This is particularly true for the size of multiplier, possible parallel processing, stages of pipelining, and algorithm strategies. In this paper, we propose an efficient hardware implementation of this primitive that can be used for asymmetric digital signature, key exchange and asymmetric encryption. To our knowledge this is the first hardware implementation of XTR.

In 1994, Smith and Skinner introduced the LUC public key cryptosystem [24] based on Lucas function. This is an analog to discrete logarithm over $\mathbb{F}_{p^2}^*$ with elements of order $p+1$ represented by their trace over $\mathbb{F}_p$. More recently, Gong and Harn [6] used a similar idea with elements in $\mathbb{F}_{p^3}^*$ of order p^2+p+1. Finally, Lentra and Verheul proposed XTR in [12], that represents elements of $\mathbb{F}_{p^6}^*$ with order (dividing) p^2-p+1 by their trace over $\mathbb{F}_{p^2}$. These representations induce security over the fields $\mathbb{F}_{p^i}^*$, with $i = 2, 3, 6$ with respect to LUC, Gong-Harn or XTR cryptosystems, whereas numbers manipulated are over $\mathbb{F}_{p^2}$ for XTR or $\mathbb{F}_p$ for the others. XTR is the most efficient out of the three since it allows a reduction factor of 3 between size of security and size of manipulated numbers.

Parameter p is chosen as a prime number. Another condition for security requirements is that there exists a sufficiently large prime number q that divides $p^2 - p + 1$. Typically, p is chosen as a 160-bit integer whereas q is a 170-bit integer. With these parameters, XTR security is considered as "equivalent" to RSA security with 1024-bit modulus or an elliptic curve cryptosystem (ECC) based on 160-bit field. The parameter p is also chosen to be equivalent to 2 modulo 3. In this case, $\mathbb{F}_{p^2}$ is isomorphic to $\mathbb{F}_p[X]/(X^2 + X + 1)$. If α denotes the root of $(X^2 + X + 1)$, then (α, α^2) is a normal basis of $\mathbb{F}_{p^2}$ over $\mathbb{F}_p$. Finally, any element of $\mathbb{F}_{p^2}$ can be represented as (x_1, x_2) with $x_1, x_2 \in \mathbb{F}_p$.

XTR operations are performed over $\mathbb{F}_{p^2}$. This is achieved by representing elements of the subgroup of $\mathbb{F}_{p^6}^*$ of order q (dividing p^2-p+1), generated by g, by their trace over $\mathbb{F}_{p^2}$. Trace over $\mathbb{F}_{p^2}$ of an element is just the sum of its conjugates. Let a be an element of $< g >$, then $\mathrm{Tr}(a) := \mathrm{Tr}_{\mathbb{F}_{p^6}/\mathbb{F}_{p^2}}(a) = a + a^{p^2} + a^{p^4}$ and $\mathrm{Tr}(a) \in \mathbb{F}_{p^2}$.

Let x and y be two elements of $\mathbb{F}_{p^2}$ represented respectively by (x_1, x_2) and (y_1, y_2), then it is shown in [12, Lem. 2.1.1] that

1. x^p is represented by (x_2, x_1) and this way computing x^p from x is obtained by permuting elements representing x,
2. x^2 is represented by $(x_2(x_2 - 2x_1), x_1(x_1 - 2x_2))$ and this way computing x^2 is done with two multiplications in $\mathbb{F}_p$,
3. $x \cdot y$ is represented by $(x_2y_2 - x_1y_2 - x_2y_1, x_1y_1 - x_1y_2 - x_2y_1)$ or by $(x_1y_1 + 2x_2y_2 - (x_1 + x_2)(y_1 + y_2), 2x_1y_1 + x_2y_2 - (x_1 + x_2)(y_1 + y_2))$ and this way

the product of two $\mathbb{F}_{p^2}$-elements is obtained through three multiplications in $\mathbb{F}_p$.
4. $x \cdot z - y \cdot z^p$ is represented by $(z_1(y_1 - x_2 - y_2) + z_2(x_2 - x_1 + y_2), z_1(x_1 - x_2 - y_1) + z_2(y_2 - x_1 + y_1))$ and this way this special operation on $\mathbb{F}_{p^2}$-elements is obtained through four multiplications in $\mathbb{F}_p$.

In the remainder of this paper, we follow the notation used in [12–15, 26]. We denote $\mathrm{Tr}(g)$ by c and for any integer k, $\mathrm{Tr}(g^k)$ by c_k. The basic operation with XTR is the analog to exponentiation, *i.e.* from an integer k and a subgroup element g of $\mathbb{F}^*_{p^6}$, computing $\mathrm{Tr}(g^k)$. This is performed in an efficient way by using formulæ from [13, Cor. 2.3.3] quoted below:

1. $c_{2n} = c_n^2 - 2c_n^p$; it is obtained with two multiplications in $\mathbb{F}_p$.
2. $c_{n+2} = c \cdot c_{n+1} - c^p \cdot c_n + c_{n-1}$; it is obtained with four multiplications in $\mathbb{F}_p$.
3. $c_{2n-1} = c_{n-1} \cdot c_n - c^p \cdot c_n^p + c_{n+1}^p$; it is obtained with four multiplications in $\mathbb{F}_p$.
4. $c_{2n+1} = c_{n+1} \cdot c_n - c \cdot c_n^p + c_{n-1}^p$; it is obtained with four multiplications in $\mathbb{F}_p$.

With the previous formulæ an XTR exponentiation is carried out using Algorithm 1.1 from [13].

Algorithm 1.1 Computation of $S_n(c)$ given n and c, from [13, Algorithm 2.3.5]

INPUT: $n = \sum_{j=0}^{r} n_j 2^j$ and c
OUTPUT: $S_n(c) = (c_{n-1},\ c_n,\ c_{n+1})$

if $n < 0$ **then** apply this algorithm to $-n$ and c, then use negative result.
if $n = 0$ **then** $S_0(c) \leftarrow (c^p, 3, c)$.
if $n = 1$ **then** $S_1(c) \leftarrow (3, c, c^2 - 2c^p)$.
if $n = 2$ **then** use formulæ from App. A and $S_1(c)$ to compute c_3.
else $\overline{S}_0(c) \leftarrow S_1(c)$ and $\overline{m} \leftarrow n$.
  if $\overline{m}$ is even **then** $\overline{m} \leftarrow \overline{m} - 1$.
  $m \leftarrow \dfrac{\overline{m} - 1}{2}$, $k = 1$,
  $\overline{S}_k(c) \leftarrow S_3(c)$ with formulæ in App. A.
  $m = \sum_{j=0}^{r} m_j 2^j$ with $m_j \in \{0, 1\}$ and $m_r = 1$.
  for j **from** $r - 1$ to 0 **do**
    if $m_j = 0$ **then** compute $\overline{S}_{2k}(c)$ from $\overline{S}_k(c)$
      (using formulæ from App. A).
    if $m_j = 1$ **then** compute $\overline{S}_{2k+1}(c)$ from $\overline{S}_k(c)$
      (using formulæ from App. A).
    $k \leftarrow 2k + m_j$
  if n is even **then** use $S_{\overline{m}}(c)$ to compute $S_{\overline{m}+1}(c)$ and $\overline{m} \leftarrow \overline{m} + 1$.
return $S_n(c) = S_{\overline{m}}(c)$

We can first remark that computing $\overline{S}_{2k}(c)$ or $\overline{S}_{2k+1}(c)$ is done exactly in the same manner. More importantly, triplet representing $\overline{S}_{2k}(c)$ and $\overline{S}_{2k+1}(c)$

can be calculated *independently*. This is one of the very useful characteristic of XTR that allows us to reach a very high speed performance in our hardware implementation.

This paper is organized as follows. Next section deals with modular product evaluation. A new algorithm, of independent interest, using a look-up table is presented together with an algorithm proposed by Koç and Hung [9]. Based on these two algorithms, Section 3 presents the main results of this paper: implementation choices and performance obtained to compute an XTR exponentiation. We also make comparison between hardware implementations of XTR and other cryptosystems like RSA and ECC. Finally, we conclude in Section 4.

2 Algorithms: Implementation Options

As already shown in Section 1.1, XTR exponentiation is done with a very uniform set of operations. Contrary to classical exponentiation where a 'square-and-multiply' algorithm is used, the only changes at each loop of XTR are the inputs. According to the bit of the exponent expressed as binary expansion, $\overline{S}_{2k}(c)$ or $\overline{S}_{2k+1}(c)$ are computed from $\overline{S}_k(c)$. Details of performed operations over $\mathbb{F}_p$ are given in Appendix A.

Costly operations are products of elements. This can be done using the Koç and Hung algorithm from [9]. An alternative is simply to use a look-up table.

2.1 Modular multiplication in hardware

Let A and B be two integers. The product of A and B cannot be achieved in one single step without a big loss in timing performance and in consumed hardware resources (area). Thus this product is usually obtained by iteratively accumulating partial products a_iB. This type of multiplier is also called *scaling accumulator* or *shift-and-add* method. One of the advantages is that only one single adder is reused for all the multiplication steps.

Unfortunately, when large numbers have to be manipulated, typically 1024-bit with RSA, the important length of the carry chain may become an issue. This is especially true when using reconfigurable hardware where the length of fast carry chains is limited to the size of columns. An alternative is the use of *redundant representations*, *i.e. carry-save representations*. This eliminates the carry propagation delay. The delay of a carry-save adder (CSA) is independent of the length of operands.

Many different algorithms to compute modular multiplication using the *shift-and-add* technique exist in the literature [2, 4, 17, 21, 23]. Most of them suggest interleaving the reduction step with the accumulating one in order to save hardware resources and computation time. The usual principle is to compute or estimate the quotient $Q = \lfloor U/p \rfloor$ and then subtract the required amount from the intermediate result.

2.2 Modular multiplication using look-up table

As aforementioned, redundant representations can lead to very good timing performances. Moreover, to obtain a light hardware, we have chosen to base the multiplication on a scaling accumulator. In order to prevent the growth in length of the temporary value of the product, the addition steps are interleaved with the reduction ones.

Let p be a prime of l bits, such as $2^{l-1} < p < 2^l$. Let A and B be two integers, $0 \leq A, B < p$. Then, the modular multiplication of A and B can simply be written as

$$\begin{aligned}
A.B \bmod p &= \Big(\sum_i (a_i.B.2^i \bmod p)\Big) \bmod p \\
&= (a_{l-1}.B.2^{l-1} \bmod p + \dots) \bmod p \\
&= \Big(((\dots(((a_{l-1}.B \bmod p).2 + a_{l-2}.B) \bmod p).2 \\
&\qquad\qquad + \dots) \bmod p).2 + a_0.B\Big) \bmod p
\end{aligned}$$

This suggests the successive reduction of the temporary value in the case of 'left-to-right' multiplication. Our fairly simple idea is based on the following observation: reduction can be carried out using a look-up table.

If S and C denote the redundant representation, the three most significant bits (MSB) of S and C are extracted and added together. The corresponding reduced number is then chosen among the precalculated values. All the $2^{3+1}-1 = 15$ possible cases are stored in memory. The reduced number is then added with the two MSB-free values, pre-multiplied by 2 before being re-used in the multiplication loop. The next partial product a_iB is also added providing a new S and C pair of redundant representation.

The operation is repeated until all bits of A have been covered. Eventually the values are processed one last time, but without new partial product input. This extra step guarantees the sum of the redundant vectors to be lower than $2p$. After the step -1, the result then requires at most one final reduction. This can be simply proven by the observation that after step 0: $S, C < 2^{l-2}$. After the shift and the addition with the feedback of the residues: $S + C < 2^l + 2p$. Since $2^l < 2p$, the following relation holds: $S + C < 4p$. Finally, dividing the result by 2 gives $R < 2p$. Algorithm 2.1 gives a detailed description.

2.3 Modular multiplication with sign estimation technique

Another type of algorithm (more advanced) was proposed by Koç and Hung in [9]. Once again, it interleaves the reduction step with the addition of the partial product and the intermediate result is stored in redundant representation.

This algorithm is based on the following clever idea: the sign of the number represented by the carry-sum pair can be evaluated and used to add/subtract a multiple of the modulus in order to keep the intermediate result within two

Algorithm 2.1 Algorithm for computing modular multiplication

INPUT: $0 < A, B < p$ and $2^{l-1} < p < 2^l$
OUTPUT: $R = AB \bmod p$

$S_l := 0$, $C_l := 0$, $a_{-1} := 0$.
for i **from** $l-1$ **to** -1 **do**
  $(S'_i, C'_i) := 2(S_{i+1} \bmod 2^{l-2}) + 2(C_{i+1} \bmod 2^{l-2}) + a_i B$.
  $(S_i, C_i) := S'_i + C'_i + 2\big[(S_{i+1} \text{ div } 2^{l-2} + C_{i+1} \text{ div } 2^{l-2}).2^{l-2} \bmod p\big]$.
$R = (S_{-1} + C_{-1})/2$.
if $R > p$ **then** $R := R - p$.
return R.

Algorithm 2.2 Algorithm from [9], computing modular multiplication

INPUT: $0 < A, B < p$ and $2^{l-1} < p < 2^l$
OUTPUT: $R = AB \bmod p$

$p' = 8p$, $S := 0$, $C := 0$
for i **from** $l-1$ **to** -3 **do** .
  if $ES(S,C) = (+)$ **then** $(S,C) := 2S + 2C + a_i B - p'$.
   else if $ES(S,C) = (-)$ **then** $(S,C) := 2S + 2C + a_i B + p'$.
   else $(S,C) := 2S + 2C + a_i B$.
  loop invariant: $S + C \in \big[-\frac{3p'}{4}, \frac{7p'}{8}\big)$.
$R := S + C$.
if $R < 0$ **then** $R := R + p'$.
return $R/8$.

boundaries. This is done by $ES(S,C)$. The sign estimation requires to add the 5 MSB of the two vectors S and C.

The skeleton is given in Algorithm 2.2 and we refer the interested reader to [9] for further details.

3 Implementation Results

3.1 Methodology

After having introduced a new algorithm for modular multiplication using look-up table in the intermediate reductions and having recalled the Koç and Hung algorithm, let us now consider the subject of this paper: XTR implementation. In this section, the global approach of the design is discussed and two architectures are presented. Implementation results and performances are given as well. Particular considerations about scalability and portability conclude the section.

One of our purposes for implementing XTR architectures on reconfigurable hardware is to achieve a well-balanced trade-off between hardware size and frequency. Nevertheless, particular care has been taken to keep the architectures open to all kinds of FPGAs. This is the reason why some available features have not been used, *e.g.* internal multipliers. This way, designs can be directly synthesized, whatever the device target.

Even if our architecture is more general than an FPGA oriented implementation, we decide to adopt the classical design methodology described in [25]. The authors introduced the concept of hardware efficiency which could be represented as the ratio *Nbr. of registers / Nbr. of LUTs*. To achieve a high level of sub-pipelining, this ratio must be as close as possible to one. This was presented in the view of designing efficient implementation of symmetric ciphers but remains partially true for general designs, at least it suggests a method. And while implementing our design, we tried to use this concept to reach high clock frequency. Implementation results appear in Section 3.

As aforementioned, the 'parallel characteristic' of XTR is obvious. Indeed, each component of $\overline{S}_n(c)$, with $n = 4k$ or $4k+1$, can be computed independently. As an illustration, if we consider elliptic curve cryptosystems point addition or doubling, many dependencies exist during computation, see [1]. This issue is removed using the Montgomery ladder principle, see for an overview [8]. Moreover each element of $\mathbb{F}^*_{p^2}$ is represented as a couple. Each component of the couple is evaluated at the same time *and* independently. Then computations for the α and α^2 components are similar and can thus be executed separately. This means that $\overline{S}_n(c)$ is represented by 6 components that can be evaluated independently. A closer look shows that the computation of c_{4k+1} (and/or c_{4k+3}) is composed of two parts alike, with a final addition. Hence it is possible to process one step of the encryption at once in parallel with *eight independent* processes.

Furthermore, operations are quite similar. A generic cell can easily be derived to design a generic process unit able to perform the encryption in a sequential mode, at a lighter hardware cost. This also underlined the flexibility of design allowed by XTR.

Parallel designs are presented underneath. The general layout of both architectures is as follows. A 160-bit shift register containing n produces the MSB m on each iteration 1.1. With respect to this bit, different multiplexors forward the data to the inputs of the corresponding processing units. Each of them computes its data and returns the results to the multiplexors, for the next iterations.

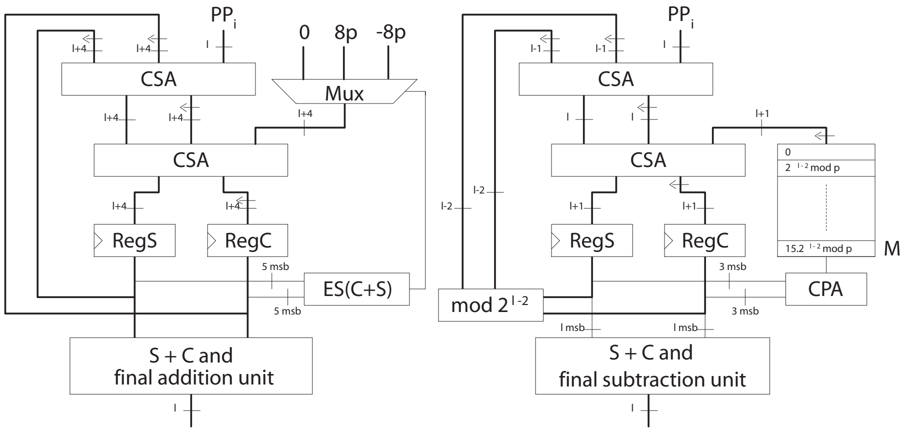
The core of the process unit is the modular multiplier. It is preceded by some logic dealing with the preliminary additions and subtractions. Its result is stored in a shift-register.

3.2 Architectures of a process unit

The internal structures of Koç and Hung algorithm and ours are displayed in Fig. 1. Our look-up table based algorithm is centered around two CSA taking as input the partial product a_iB, the $(l-2)$-bit truncated result vectors and the reduced values based on the 3 most significant bits. The originality of this method is due to the modular reduction technique. Just recall the Algorithm 2.1: the most significant bits are extracted and added together in order to keep the intermediate values in fixed boundaries. According to the initial values ($S_l = C_l = 0$), the utmost limits are $0 \leq S_i, C_i < 2^{l+1}$. The 15 possible values for

$$(S_{i+1} \text{ div } 2^{l-2} + C_{i+1} \text{ div } 2^{l-2}).2^{l-2} \bmod p \tag{1}$$

are precalculated according to p and stored in the memory (denoted M in the figure). Both 3-bit MSB are added together in order to produce 4-bit address. The memory can thus be mapped by the use of l LUT. Throughout each iteration of the multiplication, a new partial product is inserted and the feedback values must therefore be doubled.



(a) Implementation of Koç and Hung algorithm and (b) of ours.

Fig. 1. The two modular multiplication algorithms.

As previously explained, one final iteration without inserting a new partial product ensures the final result to be under $2p$. After addition by a ripple carry adder (RCA), there may thus be an extra p left over. It is easily handled by the use of another RCA and a multiplexor, as suggested in [20]. The RCA uses the fast carry chain available on every FPGA. Nevertheless the carry chain for a 170-bit RCA would lengthen the critical path. They are then composed of pipelined smaller RCAs.

The implementation structure of the Koç and Hung algorithm is very similar to ours. Most of the design choices were identical for both algorithms. The main difference lies in the number of bits taken to evaluate the estimation function (*i.e.* 2×5 for Koç and Hung algorithm and 2×3 for ours). Moreover the Koç and Hung algorithm keeps the whole value intact (no truncation is applied after the registers S and C), this requires thus a bus length of $l + 4$-bit.

3.3 Discussion about algorithms performances

Efficiency of two implementations is always difficult to compare. The same algorithm could lead to very different performances depending on the type of device used (ASIC or FPGA) and on the technology ($0.12\mu m$ CMOS for Virtex II), on the cleverness of designers (smart trade-offs between area and latency) and finally on the options chosen for *place-and-route* (PAR).

Algorithms described above present many similarities. They require two CSAs of size $O(l)$, a module of last reduction and an estimation function feeding a look-up table. Both of them shift their feedbacked S and C pair at each iteration. Koç-Hung algorithm requires less memory than ours. However, the estimation function given by Koç-Hung takes 2×5 inputs, and our algorithm takes 2×3 inputs.

A Field Programmable Gate Array is a tool situated between hardware and software. With the increase of powerful internal features it becomes very competitive compared to ASIC. We used FPGA to implement our design. This gives an advantage to our algorithm in terms of latency (critical path) with small area increase.

Most FPGA devices use 4-input look-up tables (LUTs) to implement function generators. Four independent inputs are provided into each of the 2 function generators in a slice. Then, any boolean function of four inputs can be mapped in a LUT. So the propagation delay is independent of the function implemented. Moreover, each Virtex slice contains two multiplexers (MUXF5 and MUXF6). This allows the combination of slices to obtain higher-level logic functions (any function of 5 or 6 inputs)[3].

From these considerations, we can consider the delay of the 2 estimation functions. In our algorithm, the estimation function can be mapped as a 6-input boolean function with a propagation delay of 1 LUT. In the case of Koç-Hung algorithm a 10-input function must be implemented, so 2 stages of LUT are needed. This implies a latency of 2 LUTs.

This endows to our algorithm an advantage for an FPGA implementation but the two algorithms have very similar performances and it is difficult to evaluate the performance for Koç and Hung algorithm using another technology. Table 1 gives the synthesis result of our implementation. We can see that our algorithm can achieve a higher frequency, as expected.

Our Implementation	Nbr. of LUT	Nbr. of FF.	Nbr. of Slices	Freq (MHz)	Hardware Efficiency Nbr. Reg/Nbr. LUT	AT complexity (slices*cycles/freq.)
of K-H	1,402	1,230	805	189.2	0.88	7.5 e-4
of LUT	1,450	1,246	857	203.3	0.86	7.6 e-4

Table 1. Evaluation of the performances between the two algorithms

[3] In Virtex II family, up to 8 slices can be combined with similar multiplexers.

In [3], the complexity of the implementation of a binary multiplication is formally defined. This definition includes many parameters such as the technology used, the area and time required and the length of the operand. In this way, we decide to adopt an *Area-Time Complexity* and then the product AT as an element of comparison of the algorithms we implemented.

3.4 Performances

As far as we know, this paper is the first dealing with XTR cryptosystem implementation on reconfigurable hardware. Even if it is not fully satisfactory, we decided to compare it with the best existing implementations (as we know) of the RSA algorithm [5, 16] and elliptic curve processors [18, 19]. Table 2 indicates that our implementation is definitely competitive with respect to other designs for equivalent security. Note that no assumption on the form of p has been made: this freedom brings an enormous flexibility in the use of our designs.

Implementation	Technology	Nbr. of LUT	Nbr. of FF.	Nbr. of Slices	Block RAM	Freq (MHz)	Comput. Time (ms)
RSA 1024 [5]	Xilinx XC40250XV-9	-	-	27,304	0	45.6	3.1
RSA 1024 [16]	Xilinx XC2V6000	-	-	24,767	0	100.49	2.63
ECP [18] $GF(2^m)$	Xilinx XCV400E-8	3,002	1,769	-	10	76.7	0.21
ECP [19] $GF(p)$	Xilinx XCV1000E-8	11,416	5,735	-	35	40	3
XTR with K-H	**Xilinx XC2V6000-6**	**17,903**	**13,509**	**10,607**	**0**	**150**	**0.21**
XTR with LUT	**Xilinx XC2V6000-6**	**18,103**	**13,752**	**10,737**	**0**	**162.4**	**0.21**

Table 2. Evaluation of the performances between different public-key cryptosystems. (-) denotes unknown values.

Our designs were synthesized on a Virtex2 XC2V6000-6-FF152, which contains 33,792 Slices, 144 Select RAM Blocks, 144 18-bit x 18-bit Multipliers. The synthesis was performed with Synplify Pro 7.3 (SYNPLICITY) and automatic place-and-route (PAR) was carried out with XILINX ISE 6.1i. Moreover, concerning the timing performances, we decided to pack the input/output registers of our implementation into the input/output blocks (IOB) in order to try and reach the achievable performance.

4 Conclusion

In this article, the first implementation of the XTR (crypto)system on reconfigurable hardware (FPGA) is presented. Various implementations are discussed. Evaluation of modular products is the costly part. This can be carried out using the clever algorithm from Koç and Hung. We also propose a (competitive) alternative based on look-up table. The performances of these two algorithms seem to be in a similar gap.

The main subject of this paper is XTR implementation. The intrinsic parallelism of XTR allows us to obtain a very high level of performance with very small memory requirements. Compared with RSA exponentiation, XTR appears as a very interesting alternative in hardware: an XTR exponentiation is carried out in about 0.21 ms at frequency beyond 150 MHz.

Moreover, implementations are fully generic and have been designed for any FPGA device without using any particular feature. Portability is then another characteristic of our designs. Once again there is absolutely no constraint on p (characteristic of the field over which XTR is defined). Designs are dedicated to any p up to 170 bits and it would be obvious to oversize their length. Eventually, using special forms of p (*e.g.* Mersenne primes as used for elliptic curves) could lead to considerable improvements, to the detriment of the present generality.

We stress that porting our implementation on ASIC would also underline the very good efficiency of XTR compared with RSA or elliptic curve cryptosystems.

Acknowledgments

We are grateful to Christophe Giraud and Martijn Stam for providing us fruitful comments on the preliminary version of this paper. We would also like to thank anonymous reviewers for their useful observations.

References

1. IEEE Std 1363-2000. *IEEE Standard Specifications for Public-Key Cryptography.* IEEE Computer Society, August 29, 2000.
2. Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A.M. Odlyzko, Ed., *Advances in Cryptology, Proc. Crypto86*, pages 311–323, vol. 263 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.
3. Richard P. Brent and H.T. Kung. The Area-Time Complexity of Binary Multiplication. In *J.ACM*, vol. 28, 1981, pages 521–534, July 1981.
4. Ernest F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R.L. Rivest, and A.T. Sherman, Ed., *Advances in cryptology Proc. of CRYPTO '82*, pages 51–60. Plenum Press, 1983.
5. Thomas Blum and Christof Paar. High-Radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Trans. on Computers*, 50(7), pages 759–764, 2001.

6. Guang Gong and Lein Harn. Public key cryptosystems based on cubic finite field extensions. In *IEEE Trans. on Inf. Theory*, Nov. 1999.
7. Robert Granger, Dan Page and Martijn Stam. A Comparison of CEILIDH and XTR. In D.A. Buell Ed. *Algorithmic Number Theory, 6th International Symposium – ANTS-VI*, pages 235–249, vol. 3076 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
8. Marc Joye and Sung-Ming Yen. The Montgomery powering ladder. In B.S. Kaliski Jr. and Ç. K. Koç, Ed., *Cryptographic Hardware and Embedded Systems (CHES 2002)*, pages 291–302, vol. 2523 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
9. Çetin Kaya Koç and Ching Yu Hung. A Fast Algorithm for Modular Reduction. In *IEE Proceedings - Computers and Digital Techniques*, 145(4), pages 265–271, July 1998.
10. Seongan Lim, Seungjoo Kim, Ikktwon Yie, Jaemoon Kim and Hongsub Lee. XTR Extended to GF(p^{6m}). In S. Vaudenay and A.M. Youssef, Ed., *Selected Areas in Cryptography – SAC 2001*, vol 2259 of *Lecture Notes in Computer Science*, pages 301–312. Springer-Verlag, 2001.
11. Arjen K. Lenstra. Using Cyclotomic Polynomials to Construct Efficient Discrete Logarithm Cryptosystems Over Finite Fields In V. Varadharajan, J. Pieprzyk, Y. Mu, Eds. *Information Security and Privacy, Second Australasian Conference – ACISP'97*, vol. 1270 of *Lecture Notes in Computer Science*, pages 127–138. Springer-Verlag, 1997.
12. Arjen K. Lenstra and Eric R. Verheul. The XTR public key system. In M. Bellare, Ed., *Advances in Cryptology – CRYPTO 2000*, vol. 1880 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2000.
13. Arjen K. Lenstra and Eric R. Verheul. An overview of the XTR public key system. *Public Key Cryptography and Computational Number Theory Conference*, 2000.
14. Arjen K. Lenstra and Eric R. Verheul. Key improvements to XTR. In T. Okamoto, Ed., *Advances in Cryptology – ASIACRYPT 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pages 220–233. Springer-Verlag, 2000.
15. Arjen K. Lenstra and Eric R. Verheul. Fast irreductibility and subgroup membership testing in XTR. In K. Kim, Ed., *Public Key Cryptography – PKC 2001*, vol. 1992 of *Lecture Notes in Computer Science*, pages 73–86. Springer-Verlag, 2001.
16. Ciaran McIvor, Máire McLoone, John McCanny, Alan Daly and William Marnane. Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures. *37th Asilomar Conference on Signals, Systems, and Computers*, Nov. 2003.
17. Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170), pages 519–521, April 1985.
18. Gerardo Orlando and Christof Paar. A High Performance Reconfigurable Elliptic Curve Processor for $GF(2m)$. In Ç.K. Koç, C. Paar, Ed., *Cryptographic Hardware and Embedded Systems (CHES 2000)*, vol. 1965 of *Lecture Notes in Computer Science*, pages 41–56, Springer-Verlag, 2001.
19. Gerardo Orlando and Christof Paar. A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware. In Ç.K. Koç, D. Naccache, C. Paar, Ed., *Cryptographic Hardware and Embedded Systems (CHES 2001)*, vol. 2162 of *Lecture Notes in Computer Science*, pages 348–363, Springer-Verlag, 2001.
20. Behrooz Parhami. RNS representation with redundant residues. In Proc. of the *35th Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, pages 1651-1655, November 4-7, 2001.

21. Jean-Jacques Quisquater. Fast modular exponentiation without division. At *Rump session of EUROCRYPT '90*, May 1990.
22. Karl Rubin and Alice Silverberg. Torus-based cryptography. In D. Boneh, Ed., *Advances in Cryptology – CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer*, pages 349–365. Springer, 2003.
23. Holger Sedlak. The RSA cryptography processor. In D. Chaum and W.L. Price, Ed., *Advances in Cryptology - EUROCRYPT '87*, Amsterdam, The Netherlands, vol. 304 of *Lecture Notes in Computer Science*, pages 95–105. Springer-Verlag, 1988.
24. Peter Smith and Christopher Skinner. A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discret logarithms. In J. Pieprzyck and R. Safavi-Naini, Ed., *Advances in Cryptology – ASIACRYPT '94*, vol. 917 of *Lecture Notes in Computer Science*, pages 357–364. Springer-Verlag, 1995.
25. François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, Jean-Didier Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In C. Walter, Ed., *Cryptographic Hardware and Embedded Systems (CHES 2003)*, Volume 2779 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2003.
26. Martijn Stam and Arjen K. Lenstra. Speeding up XTR. In C. Boyd, Ed., *Advances in Cryptology – ASIACRYPT 2001*, vol. 2248 of *Lecture Notes in Computer Science*, pages 125–143. Springer-Verlag, 2001.
27. Eric R. Verheul. Evidence that XTR Is More Secure then Supersingular Elliptic Curve Cryptosystems. In B. Pfitzmann, Ed., *Advances in Cryptology – EUROCRYPT 2003*, vol. 2045 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag, 2001.
28. Johannes Wolkerstorfer. Dual-Field Arithmetic Unit for $GF(p)$ and $GF(2^m)$. In B.S. Kaliski Jr., Ç.K. Koç, C. Paar, Ed., *Cryptographic Hardware and Embedded Systems CHES 2002*, vol. 2523 of *Lecture Notes in Computer Science*, pages 500–514. Springer-Verlag, 2002.

A Details of Basic XTR Operations

To compute $S_{2k} = (c_{4k}, c_{4k+1}, c_{4k+2})$ from $S_k(c_{2k}, c_{2k+1}, c_{2k+2})$, the following operations are required:

$$\begin{aligned}
c_{4k} &= c_{2k}^2 - 2c_{2k}^p \\
&= \big(c_{2k,2}(c_{2k,2} - 2c_{2k,1} - 2)\big)\alpha + \big(c_{2k,1}(c_{2k,1} - 2c_{2k,2} - 2)\big)\alpha^2 \\
c_{4k+1} &= c_{2(2k+1)-1} = c_{2k}c_{2k+1} - c^p c_{2k+1}^p + c_{2k+2}^p \\
&= \big(c_{2k+1,1}(c_{1,2} - c_{2k,2} - c_{1,1}) + c_{2k+1,2}(c_{2k,2} - c_{2k,1} + c_{1,1}) + c_{2k+2,2}\big)\alpha \\
&\qquad + \big(c_{2k+1,1}(c_{2k,1} - c_{2k,2} + c_{1,2}) + c_{2k+1,2}(c_{1,1} - c_{2k,1} - c_{1,2}) + c_{2k+2,1}\big)\alpha^2 \\
c_{4k+2} &= c_{2k+1}^2 - 2c_{2k+1}^p \\
&= \big(c_{2k+1,2}(c_{2k+1,2} - 2c_{2k+1,1} - 2)\big)\alpha + \big(c_{2k+1,1}(c_{2k+1,1} - 2c_{2k+1,2} - 2)\big)\alpha^2
\end{aligned}$$

Computing $S_{2k+1} = (c_{4k+2}, c_{4k+3}, c_{4k+4})$ from $S_k(c_{2k}, c_{2k+1}, c_{2k+2})$, is done with the following operations:

$$
\begin{aligned}
c_{4k+2} &= c_{2k+1}^2 - 2c_{2k+1}^p \\
&= \big(c_{2k+1,2}(c_{2k+1,2} - 2c_{2k+1,1} - 2)\big)\alpha + \big(c_{2k+1,1}(c_{2k+1,1} - 2c_{2k+1,2} - 2)\big)\alpha^2 \\
c_{4k+3} &= c_{2(2k+1)+1} = c_{2k+2}c_{2k+1} - cc_{2k+1}^p + c_{2k}^p \\
&= \big(c_{2k+1,1}(c_{1,1} - c_{2k+2,2} - c_{1,2}) + c_{2k+1,2}(c_{2k+2,2} - c_{2k+2,1} + c_{1,2}) + c_{2k,2}\big)\alpha \\
&\quad + \big(c_{2k+1,1}(c_{2k+2,1} - c_{2k+2,2} + c_{1,1}) + c_{2k+1,2}(c_{1,2} - c_{2k+2,1} - c_{1,1}) + c_{2k,1}\big)\alpha^2 \\
c_{4k+4} &= c_{2k+2}^2 - 2c_{2k+2}^p \\
&= \big(c_{2k+2,2}(c_{2k+2,2} - 2c_{2k+2,1} - 2)\big)\alpha + \big(c_{2k+2,1}(c_{2k+2,1} - 2c_{2k+2,2} - 2)\big)\alpha^2
\end{aligned}
$$