On Repeated Squarings in Binary Fields

Kimmo U. Järvinen

Helsinki University of Technology (TKK)
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
kimmo.jarvinen@tkk.fi

Abstract. In this paper, we discuss the problem of computing repeated squarings (exponentiations to a power of 2) in finite fields with polynomial basis. Repeated squarings have importance, especially, in elliptic curve cryptography where they are used in computing inversions in the field and scalar multiplications on Koblitz curves. We explore the problem specifically from the perspective of efficient implementation using field-programmable gate arrays (FPGAs) where the look-up table (LUT) structure helps to reduce both area and delay overheads. In fact, we show that the optimum construction depends on the size of the LUTs. We propose several repeated squarer architectures and demonstrate their practicability for FPGA-based implementations. Finally, we show that the proposed repeated squarers can offer significant speedups and even improve resistivity against side-channel attacks.

1 Introduction

Squaring is the operation where an element is multiplied by itself. We explore the problem of computing *repeated squarings*, i.e. several successive squarings, in finite binary fields, $\mathbb{F}_{2^m}$, with polynomial basis and present hardware solutions for it. In this paper, we consider repeated squarings mainly in the context of elliptic curve cryptography [1,2], but generalizations to other application domains are straightforward.

Repeated squarings have two important applications in elliptic curve cryptography:

1. *Itoh-Tsujii inversion* [3] is a method based on Fermat's Little Theorem that finds the multiplicative inverse of $a \in \mathbb{F}_{2^m}$ by efficiently computing a^{2^m-2}. In this exponentiation the number of squarings is considerably higher than the number of multiplications and, hence, it includes many repeated squarings. Originally, Itoh-Tsujii inversion was proposed for binary fields over normal basis where (repeated) squarings are trivial, but it is a viable solution also for polynomial basis [4].
2. *Koblitz curves* [5] are a class of elliptic curves over $\mathbb{F}_{2^m}$, where fast Frobenius maps can be used instead of computationally more demanding point doublings resulting in considerably faster computations. Frobenius is computed by squaring the coordinates of a point on the curve; thus, successive Frobenius maps require a repeated squaring for each coordinate.

M.J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini (Eds.): SAC 2009, LNCS 5867, pp. 331–349, 2009.

Recent studies show that when the above cases are implemented in hardware, a considerable portion of the total computation time is consumed in squarings [6,7]; hence, they motivate the research on efficient computation of repeated squarings.

Field-programmable gate arrays (FPGAs) are popular implementation platforms for cryptographic algorithms because their combination of speed and programmability offers many advantages over general-purpose processors and application specific circuits [8]. The basic building blocks of an FPGA are n-to-1 bit *look-up tables* (n-LUTs). The most typical LUT size is $n = 4$, but larger n are used in some contemporary FPGA architectures. Despite the vast amount of papers describing FPGA-based cryptographic implementations, the effects of LUT size have been studied surprisingly little. Papers introducing finite field arithmetic units, e.g. multipliers, typically consider only 2-to-1 bit gates. In this paper, we demonstrate how optimizations for a specific LUT structure can considerably improve finite field arithmetic units.

There are only few studies on hardware implementation of repeated squarings. Lutz and Hasan [9] showed how repeated squarings can be accelerated by attaching a register directly after a squarer and feeding its content back to the input of the squarer. This removed the need of storing intermediate values outside the squarer and, as a result, computing e repeated squarings required only e clock cycles [9]. Similar repeated squarers were later used by Järvinen and Skyttä [6]. Recently, Rebeiro and Mukhopadhyay [7] observed that quading, i.e. a^4, is only slightly more expensive than squaring in FPGAs. They utilized this observation by building a repeated squarer using a chain of quadings and used it for accelerating Itoh-Tsujii inversions.

This paper contributes in the following ways:

- We provide simple tools for analyzing repeated squarings in binary fields defined by arbitrary irreducible polynomials and use them for analyzing repeated squarings in fields defined by NIST in [10];
- We generalize the observation of [7] and demonstrate how LUTs can be exploited in designing efficient circuitries for repeated squarings;
- We develop efficient repeated squarers with (a) fixed exponents and (b) varying exponents and provide implementation results on Xilinx FPGAs (Spartan-3A 3S1400-5 and Virtex-5 5VLX50-3) demonstrating the practicability of the repeated squarers; and
- We show how repeated squarings can improve speed of Itoh-Tsujii inversions (or, more generally, exponentiations in binary fields) and elliptic curve cryptography on Koblitz curves and even increase resistivity against side-channel attacks.

We begin by introducing the preliminaries of repeated squarings in Sec. 2. In Sec. 3, we analyze repeated squarings with a focus on fields defined by NIST in [10]. Based on the results of the analysis, we design repeated squarers with both a fixed exponent and a varying exponent in Sec. 4 and present results on Xilinx FPGAs in Sec 5. We end with conclusions and possible directions for future research in Sec. 6.

2 Squaring in Binary Fields

A *binary field*, $\mathbb{F}_{2^m}$, is generated from the ring of polynomials over $\mathbb{F}_2$, $\mathbb{F}_2[x]$, with an *irreducible polynomial*, $p(x)$, with a degree m by setting $\mathbb{F}_{2^m} : \mathbb{F}_2[x]/p(x)$. In *polynomial basis*, an element of $\mathbb{F}_{2^m}$ is represented as a binary polynomial with a degree at most $m-1$, i.e.,

$$a(x) = \sum_{i=0}^{m-1} a_i x^i. \tag{1}$$

Operations in polynomial basis are computed modulo $p(x)$. Addition, $a(x)+b(x)$, is simply a bitwise exclusive-or (xor). Multiplication, $a(x) \times b(x)$, is more complicated and divides into two steps: (1) multiplication in $\mathbb{F}_2[x]$ and (2) reduction modulo $p(x)$. *Squaring* is a special case of multiplication where $a(x) = b(x)$. For squaring the first step of multiplication is performed simply by inserting zeros between each bit in the bitvector representing $a(x)$. Thus, squaring becomes

$$a^2(x) = \sum_{i=0}^{m-1} a_i x^{2i} \bmod p(x) \tag{2}$$

and it essentially requires only the reduction part of the multiplication (if implemented in hardware). The reduction is computed by taking the remainder after a division with $p(x)$ in $\mathbb{F}_2[x]$.

Squaring, $b(x) = a^2(x)$, can be seen as a linear transformation described by the matrix multiplication $\mathbf{b} = \mathbf{Q}\mathbf{a}$ where $\mathbf{a} = \begin{bmatrix} a_0 a_1 \cdots a_{m-1} \end{bmatrix}^T$ represents $a(x)$, $\mathbf{b} = \begin{bmatrix} b_0 b_1 \cdots b_{m-1} \end{bmatrix}^T$ represents the result $b(x)$, and the $m \times m$ matrix $\mathbf{Q}$ with elements in $\mathbb{F}_2$ is given by [4]:

$$\mathbf{Q} = \begin{bmatrix} 1 & q_{0,1} & q_{0,2} & \cdots & q_{0,m-1} \\ 0 & q_{1,1} & q_{1,2} & \cdots & q_{1,m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & q_{m-1,1} & q_{m-1,2} & \cdots & q_{m-1,m-1} \end{bmatrix}. \tag{3}$$

See, e.g., [4] for description of the calculation of the coefficients $q_{i,j} \in \mathbb{F}_2$. A repeated squaring, $b(x) = a^{2^e}(x)$ with $e \geq 1$, is given by: $\mathbf{b} = \mathbf{Q}^e\mathbf{a}$ [4,11].

Using *normal basis* is another way of representing the elements of $\mathbb{F}_{2^m}$. A normal basis is constructed by taking a normal element in $\mathbb{F}_{2^m}$, i.e., an element $\alpha \in \mathbb{F}_{2^m}$ for which $\alpha, \alpha^2, \alpha^{2^2}, \ldots, \alpha^{2^{m-1}}$ are linearly independent. Then, an element $a \in \mathbb{F}_{2^m}$ is represented by

$$a = \sum_{i=0}^{m-1} a_i \alpha^{2^i} \tag{4}$$

Because $\alpha^{2^m} = \alpha$, squaring in normal basis is simply a rotation of the bitvector and, as a consequence, essentially free in hardware. Similarly, a repeated squaring

is free if e is fixed (rotation by e bits). A repeated squaring in normal basis can be described with a squaring matrix defined by $q_{i,j} = 1$ if $i \equiv (j + e) \bmod m$, else $q_{i,j} = 0$. Despite the efficiency of (repeated) squarings in hardware, normal bases are less frequently used in contemporary cryptosystems than polynomial bases. The main reasons are their inefficiency on software and the complexity of multiplications. It should be also noted that if e varies, then squaring is not free in normal basis either, but still cheaper than in polynomial basis.

2.1 Inversion with Fermat's Little Theorem

The element $b \in \mathbb{F}_q$ is the multiplicative inverse of an element $a \in \mathbb{F}_q$ if they satisfy $a \times b = 1$. *Inversion* is a common problem in both cryptography and codes. There are essentially two ways to compute it: Extended Euclidean Algorithm and Fermat's Little Theorem. In this paper, we discuss the latter one.

Fermat's Little Theorem states that $a^p \equiv a \pmod p$ and it follows that:

$$a^{-1} = a^{q-2} \tag{5}$$

for all $a \in \mathbb{F}_q$. Hence, an inversion in $\mathbb{F}_{2^m}$ is an exponentiation to the power $2^m - 2$. Because $2^m - 2 = \langle 11 \ldots 1110 \rangle$, the *binary method* (see, e.g, [12]) requires $m-2$ multiplications and $m-1$ squarings, none of which are repeated squarings. A more efficient method, here referred to as Itoh-Tsujii inversion, was proposed in [3] requiring $\lfloor \log_2(m-1) \rfloor + w(m-1) - 1$ multiplications and $m-1$ squarings where $w(\cdot)$ is the *Hamming weight*, i.e., the number of nonzeros in the expansion. Itoh-Tsujii inversion requires only $\lfloor \log_2(m-1) \rfloor + w(m-1)$ repeated squarings [4].

2.2 Elliptic Curve Cryptography

Elliptic curve cryptosystems [1,2] are build around an operation called *scalar multiplication*, kP, where P is a point on the curve and k is an integer. Scalar multiplication is computed with algorithms that are analogous to exponentiation algorithms (see, e.g., [12] for a review) with the exception that multiplications are replaced with operations on the curve. For instance, the binary method for exponentiation has the following analogue on an elliptic curve: k is represented using binary expansion and scanned one bit, k_i, at a time. Each bit requires a point doubling and $k_i = 1$ yields a point addition. An ℓ-bit k hence requires ℓ point doublings and $w(k)$ point additions.

Koblitz curves [5] are appealing because point doublings can be replaced with cheap *Frobenius endomorphisms* which map a point (x, y) to the point (x^2, y^2). Thus, on Koblitz curves a string of $e-1$ zeros in the expansion of k results in e consecutive Frobenius maps which are performed by computing (x^{2^e}, y^{2^e}), i.e., with two repeated squarings. A common way to compute scalar multiplications on Koblitz curves is to represent k in a width-ω τ-adic non-adjacent form (τNAF) with $w(k) \approx m/(\omega+1)$ so that there are no adjacent nonzeros [13]. For example, if k (or part of k) in width-2 τNAF is $\langle 100\bar{1}0001 \rangle$, it results in (from left to right) point addition, two repeated squarings (x^{2^3} and y^{2^3}), point subtraction, two repeated squarings (x^{2^4} and y^{2^4}), and point addition. Hence, the total cost is

$w(k)$ point additions(/subtractions) and $2w(k)$ or $2(w(k)-1)$ repeated squarings depending on whether the last digit is zero or nonzero, respectively. Points are often represented in projective coordinates, e.g. [14], and, in that case, Frobenius involves three squarings changing the costs accordingly. Repeated squarings can be useful also on general curves because scalar multiplication always requires at least one inversion.

The binary method is inherently vulnerable to *power analysis side-channel attacks* [15] because different operations are performed depending on whether a bit is zero or nonzero. Hasan [16] noted that using repeated squarings in normal basis is an efficient countermeasure because then the number of consecutive Frobenius maps, and thus the number of consecutive zeros in k, is not distinguishable with the power analysis. In polynomial basis, however, such solutions have not existed prior to this paper.

3 Analysis of Repeated Squarings

We begin with definitions and theorems which are used in analyzing implementation aspects of repeated squarings.

Definition 1. *Weight,* $\mathcal{W}(\mathbf{Q}^e)$*, is the number of ones in* $\mathbf{Q}^e$*.*

Fig. 1 plots $\mathcal{W}(\mathbf{Q}^e)$ with $1 \leq e \leq 25$ for all fields defined by NIST in [10]. Fig. 1 shows that repeated squarings with small(ish) exponents, e, have small weights, but when e grows, the weights quickly become approximately $m(m-1)/2$ [11]. The benefits gained from the sparseness of irreducible polynomials are clearly visible in the figure: the weights increase considerably slower for the fields defined by trinomials ($\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$) than pentanomials ($\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{283}}$, and $\mathbb{F}_{2^{571}}$). In the following, we focus on the NIST field, arguably, having the most contemporary relevance, $\mathbb{F}_{2^{233}} : \mathbb{F}_2[x]/x^{233} + x^{74} + 1$, but the results and conclusions that follow are easy to generalize also for other NIST fields (or to any field defined by a sparse irreducible polynomial).

While the weight gives some insight into the actual cost, it is far too simplistic for our purposes. For example, it does not take the characteristics of an implementation platform into account, and neither does it say anything about the delay of the computation, both of which are highly relevant for an implementor.

Definition 2. *Row-weight,* $\mathcal{W}_i(\mathbf{Q}^e)$*, is the number of ones on the* i^{th} *row of* $\mathbf{Q}^e$*.*

The row-weights reflect the costs of computing individual bits of the result. The row-weight is a valuable tool for computing the cost of a repeated squaring circuitry, as will be shown in the following.

Definition 3. *Area,* $\mathcal{A}_n(\mathbf{Q}^e)$*, is the number of n-LUTs required to implement* $\mathbf{Q}^e$*.*

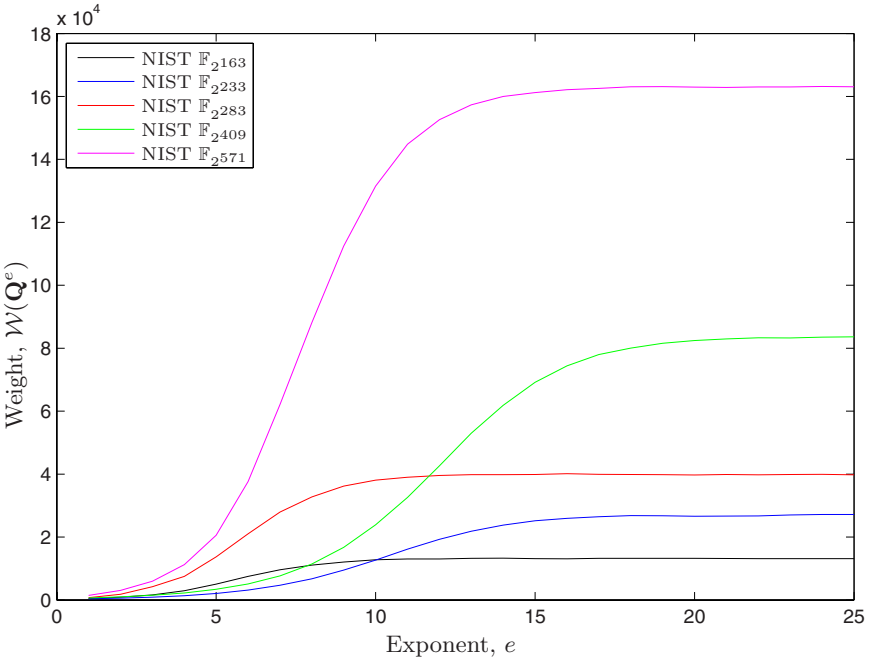

Fig. 1. The weights $\mathcal{W}(\mathbf{Q}^e)$ for the fields defined by NIST

Theorem 1. *It is possible to implement* $\mathbf{Q}^e$ *with a circuit whose area* $\mathcal{A}_n(\mathbf{Q}^e)$ *satisfies*

$$\mathcal{A}_n(\mathbf{Q}^e) \leq \sum_{i=1}^{m} \left\lceil \frac{\mathcal{W}_i(\mathbf{Q}^e) - 1}{n-1} \right\rceil . \tag{6}$$

Proof. (Sketch) Let L be the number of n-LUTs in a tree computing xor of its inputs. The number of inputs in the tree is $L(n-1)+1$ (If $L = 1$, all n inputs of the LUT are available. After that, the output of each new LUT consumes one of the existing inputs, thus, increasing the number of inputs by only $n-1$). Because computing the i^{th} row of $\mathbf{Q}^e$ requires a xor of $\mathcal{W}_i(\mathbf{Q}^e)$ bits, it follows directly from the previous that the number of LUTs is given by

$$L = \left\lceil \frac{\mathcal{W}_i(\mathbf{Q}^e) - 1}{n-1} \right\rceil . \tag{7}$$

Summing over all rows gives the upper bound of (6). It might be possible to share resources between rows and, as a result, produce an even smaller circuit; hence, the inequality. □

Definition 4. *Critical path,* $\mathcal{D}_n(\mathbf{Q}^e)$*, is the length of the longest path of consecutive* n*-LUTs in the circuit computing* $\mathbf{Q}^e$*.*

Theorem 2. *Critical path, $\mathcal{D}_n(\mathbf{Q}^e)$, is bounded by*

$$\mathcal{D}_n(\mathbf{Q}^e) \leq \max_i \lceil \log_n \mathcal{W}_i(\mathbf{Q}^e) \rceil. \tag{8}$$

Proof. (Sketch) As mentioned in the proof of Theorem 1, the tree corresponding to the i^{th} row of $\mathbf{Q}^e$ must have $\mathcal{W}_i(\mathbf{Q}^e)$ inputs and it is easy to show that the tree has the depth $\lceil \log_n \mathcal{W}_i(\mathbf{Q}^e) \rceil$. The maximum is taken because, by Definition 4, the critical path is the longest path required by the repeated squaring. Again, resource sharing may enable even shorter critical paths. □

Example 1. Consider repeated squaring in $\mathbb{F}_2[x]/x^4 + x + 1$. Let us analyze the case $e = 2$ which gives us the following repeated squaring matrix:

$$\mathbf{Q}^2 = \begin{bmatrix} 1\,1\,1\,1 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,0\,0\,1 \end{bmatrix}. \tag{9}$$

Clearly, we have $\mathcal{W}(\mathbf{Q}^2) = 9$ and $\mathcal{W}_1(\mathbf{Q}^2) = 4$, $\mathcal{W}_2(\mathbf{Q}^2) = 2$, $\mathcal{W}_3(\mathbf{Q}^2) = 2$, and $\mathcal{W}_4(\mathbf{Q}^2) = 1$. Using (6) we find out that the number of 2-LUTs (xor gates) required in implementation is bounded by $\mathcal{A}_2(\mathbf{Q}^2) \leq 5$. Indeed, we can reduce the cost to $\mathcal{A}_2(\mathbf{Q}^2) = 4$ by reusing either the xor of the 2^{nd} row or the 3^{rd} row on the 1^{st} row. With 4-LUTs used in many FPGAs, we have the minimum cost: $\mathcal{A}_4(\mathbf{Q}^2) = 3$. (8) gives the critical paths $\mathcal{D}_2(\mathbf{Q}^2) = 2$ and $\mathcal{D}_4(\mathbf{Q}^2) = 1$.

Example 2. Table 1 shows the upper bounds of (6) and (8) for NIST $\mathbb{F}_{2^{233}}$ with $n \in [2, 7]$ and $e \in [1, 6]$. Squaring ($e = 1$) in NIST $\mathbb{F}_{2^{233}}$ requires 153 LUTs with all n. The benefits of larger n are clear: e.g., if $n = 2$, repeated squaring with $e = 4$ consumes 7.5 times more area and 4.0 times more delay than a squaring but, if $n = 7$, the differences are only 1.9 and 2.0 times[1].

Table 1. Area and delay estimates given by (6) and (8) for $\mathbb{F}_{2^{233}}$

	$\mathcal{A}_n(\mathbf{Q}^e)$						$\mathcal{D}_n(\mathbf{Q}^e)$					
	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$
$e=1$	153	153	153	153	153	153	1	1	1	1	1	1
$e=2$	361	245	230	230	230	230	2	2	2	1	1	1
$e=3$	676	385	349	238	233	233	3	2	2	2	1	1
$e=4$	1141	616	466	358	349	291	4	3	2	2	2	2
$e=5$	1844	973	699	550	466	396	4	3	2	2	2	2
$e=6$	2892	1511	1035	812	663	580	5	3	3	2	2	2

Next, we show how to implement efficient repeated squarers by exploiting the fact that repeated squarings are cheap with small e (as demonstrated in Table 1).

[1] Notice that this does not imply that the time required in computation with $n = 7$ is shorter than with $n = 2$ because the delays of LUTs with different n are not the same; i.e., a LUT with small n is probably faster than one with large n.

4 Architectures for Repeated Squarers

4.1 Fixed Exponent

First, we consider the case where the exponent e is fixed; i.e, the problem is to produce a circuitry that computes only $a^{2^e}(x)$ optimally with respect to some metric, such as, area, delay, or area-delay product.

There are two simple approaches to compute a repeated squaring with e:

1. *Direct* where one produces a circuitry directly from the matrix $\mathbf{Q}^e$; and
2. *Square chain* where one produces a combinatorial circuitry of e successive squarings[2].

Let $||$ denote a *concatenation* of two circuits, i.e., the outputs of the circuit on the left are fed to the inputs of the circuit on the right. Because $a^{2^e}(x) = (a^{2^{e_1}})^{2^{e_2}}(x)$ if $e = e_1 + e_2$, a repeated squaring with an exponent e can be implemented with a chain $\mathbf{Q}^{e_1}||\mathbf{Q}^{e_2}||\ldots||\mathbf{Q}^{e_N}$ where $\sum_{i=1}^{N} e_i = e$. The direct approach is the special case where $N = 1$ and $e_1 = e$ and the square chain is the special case where $N = e$ and $e_i = 1$ for all i.

Example 3. $\mathbf{Q}^3||\mathbf{Q}^2$ denotes a circuit that given an input $a(x)$, first, computes $a^{2^3}(x)$ with a direct approach using $\mathbf{Q}^3$ and then feeds its result $b(x)$ to a circuit that computes $b^{2^2}(x)$ with $\mathbf{Q}^2$. Hence, the result from the entire circuit is $a^{2^5}(x)$.

Clearly, area and delay of concatenated circuits are bounded by

$$\mathcal{A}_n(\mathbf{Q}^{e_1}||\ldots||\mathbf{Q}^{e_N}) \leq \sum_{i=1}^{N} \mathcal{A}_n(\mathbf{Q}^{e_i}) \quad \text{and} \tag{10}$$

$$\mathcal{D}_n(\mathbf{Q}^{e_1}||\ldots||\mathbf{Q}^{e_N}) \leq \sum_{i=1}^{N} \mathcal{D}_n(\mathbf{Q}^{e_i}). \tag{11}$$

The upper bounds reflect the case where the circuits are concatenated as such without any optimizations between blocks. In practice, optimizations between concatenated circuits can reduce both area and delay. Nevertheless, in the following analysis, we assume the worst case, i.e., the equality in (10) and (11). We also assume the following ordering for the exponents: $e_1 \geq e_2 \geq \ldots \geq e_N$.

Remark 1. Although synthesis usually manages to perform optimizations so that the resulting area and delay satisfy the bounds of (10) and (11), it is also possible that the synthesis fails and results in a larger area and/or delay. Even in that case, the above bounds are always achievable by preventing synthesis from performing optimizations between blocks.

Next, setups optimized for delay, area, and their product are discussed in more detail. In all cases, the task is to find a concatenation $\mathbf{Q}^{e_1}||\mathbf{Q}^{e_2}||\ldots||\mathbf{Q}^{e_N}$ that minimizes the metric under optimization.

[2] Notice that this is different from the repeated squarer of [9] which iterates a squarer for e clock cycles.

Minimum Area. The task is to find $\{e_1, e_2, \ldots, e_N\}$ with $e = \sum_{i=1}^{N} e_i$ that minimizes $\sum_{i=1}^{N} \mathcal{A}_n(\mathbf{Q}^{e_i})$.

Using only two inputs of an n-LUT costs as much as using all n inputs. As a consequence, even though $\mathcal{W}(\mathbf{Q}^e)$ grows rapidly when e increases (see Fig. 1), taking unused inputs in use attenuates the growth of the number of LUTs. The number of LUTs hence grows only moderately with small e as shown in Table 1. In the following, the goal is to utilize the region of moderate growth also for large e by using concatenations.

In order to design a repeated squarer for a large e with minimal area, it is critical to minimize the area used per squaring. We call the exponent, $\hat{e}$, that minimizes $\mathcal{A}_n(\mathbf{Q}^{\hat{e}})/\hat{e}$ with $\hat{e} \leq e$ the *optimal exponent*, e_{opt}.

Example 4. Fig. 2 plots $\mathcal{A}_n(\mathbf{Q}^{\hat{e}})/\hat{e}$ for NIST $\mathbb{F}_{2^{233}}$ with $2 \leq n \leq 7$ and $\hat{e} \leq 8$. It shows that the number of LUTs required per squaring first decreases if $n > 2$. Consider the case $n = 4$ (e.g., Spartan 3-A) and $e = 6$. Then, $e_{\text{opt}} = 2$ and it is more area efficient to use a concatenation $\mathbf{Q}^2||\mathbf{Q}^2||\mathbf{Q}^2$ than any other concatenation or direct $\mathbf{Q}^6$ (actually, direct $\mathbf{Q}^6$ requires the largest area because $\mathcal{A}_4(\mathbf{Q}^6)/6 > \mathcal{A}_4(\mathbf{Q}^{\hat{e}})/\hat{e}$ for all $1 \leq \hat{e} < 6$).

The phenomenon of Example 4 happens for all fields defined by NIST in [10], but the benefits are expectedly larger for the fields defined by trinomials. Notice that

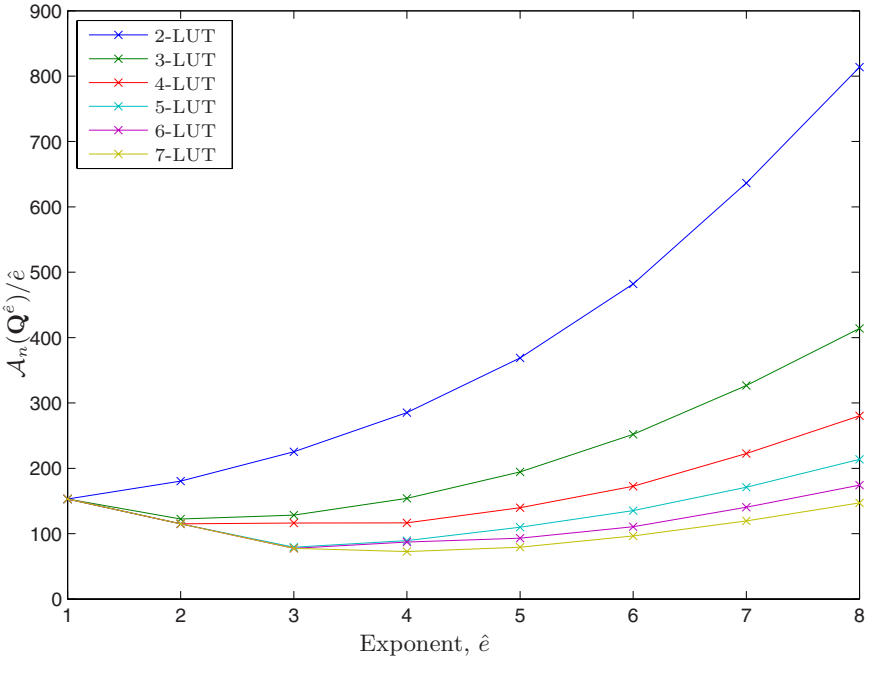

Fig. 2. Area per exponent, $\mathcal{A}_n(\mathbf{Q}^{\hat{e}})/\hat{e}$, with different LUT sizes for NIST $\mathbb{F}_{2^{233}}$

because $\mathcal{A}_2(\mathbf{Q}^{\hat{e}})/\hat{e}$ is strictly increasing, the phenomenon remains unobserved if one considers only 2-to-1 bit gates (2-LUTs).

A concatenation with minimal area can be found with an *exhaustive search* through all possible concatenations. Another approach to find a concatenation with a minimal or near-minimal area is a straightforward application of the *greedy algorithm* which results in a concatenation of $t = \lfloor e/e_{\text{opt}} \rfloor$ instances of $\mathbf{Q}^{e_{\text{opt}}}$ followed by $\mathbf{Q}^{e-te_{\text{opt}}}$ if $e_{\text{opt}} \nmid e$. The greedy algorithm guarantees an optimal concatenation if $e_{\text{opt}} \mid e$. However, if $e_{\text{opt}} \nmid e$, a concatenation with $\hat{e} > e_{\text{opt}}$ may result in a smaller area. Obviously, the greedy algorithm is computationally much simpler than the exhaustive search, especially, with large e; however, the computational complexity of the exhaustive search is insignificant if it is performed offline, as it typically is.

Example 5. Consider using 6-LUTs (e.g., Virtex-5) for implementing a repeated squarer for NIST $\mathbb{F}_{2^{233}}$ with a fixed exponent $e = 9$. Fig. 2 shows that $e_{\text{opt}} = 3$. Because $3 \mid 9$, both exhaustive search and the greedy algorithm return the same optimal concatenation: $\mathbf{Q}^3||\mathbf{Q}^3||\mathbf{Q}^3$, which has an area estimate of 699 LUTs. However, if $e = 10$, exhaustive search gives $\mathbf{Q}^4||\mathbf{Q}^3||\mathbf{Q}^3$ with an area estimate of 757 LUTs but, because $3 \nmid 10$, the greedy algorithm fails to find the optimal concatenation and returns $\mathbf{Q}^3||\mathbf{Q}^3||\mathbf{Q}^3||\mathbf{Q}$ with an estimated area of 852 LUTs.

Minimum Delay. The task is to find $\{e_1, e_2, \ldots, e_N\}$ with $e = \sum_{i=1}^{N} e_i$ that minimizes $\sum_{i=1}^{N} \mathcal{D}_n(\mathbf{Q}^{e_i})$.

Because delay grows logarithmically to $\mathcal{W}_i(\mathbf{Q}^e)$ as shown in (8), it is clear that $\mathcal{D}_n(\mathbf{Q}^e) \leq \mathcal{D}_n(\mathbf{Q}^{e_1}|| \ldots ||\mathbf{Q}^{e_N})$, and the minimum delay is always achieved with a direct circuit for $\mathbf{Q}^e$.

Minimum Area-Delay Product. The task is to find $\{e_1, e_2, \ldots, e_N\}$ with $e = \sum_{i=1}^{N} e_i$ that minimizes $\left(\sum_{i=1}^{N} \mathcal{A}_n(\mathbf{Q}^{e_i})\right)\left(\sum_{i=1}^{N} \mathcal{D}_n(\mathbf{Q}^{e_i})\right)$.

These optimizations are analogous to the minimum area optimizations; rather than minimizing $\mathcal{A}_n(\mathbf{Q}^e)$, one minimizes the product $\mathcal{A}_n(\mathbf{Q}^e)\mathcal{D}_n(\mathbf{Q}^e)$. Hence, we omit further analysis.

4.2 Varying Exponent

In Sec. 4.1, e was assumed fixed. However, many practical applications require support for varying e and, in the following, we discuss two solutions for providing such a support. The *first solution* is a simple generalization of the fixed exponent repeated squarer and suits for cases requiring support for distinct exponents. The *second solution* targets to situations where support for all exponents in a certain range is needed (for simplicity and without any loss of generality[3], we assume that the range is $0 \leq e \leq e_{\max}$).

[3] The general case, $e_{\min} \leq e \leq e_{\max}$, can be realized by using a fixed exponent squarer with $e_{\min}$ to reach the lower bound and then the second solution for the range $0 \leq e \leq e_{\max} - e_{\min}$.

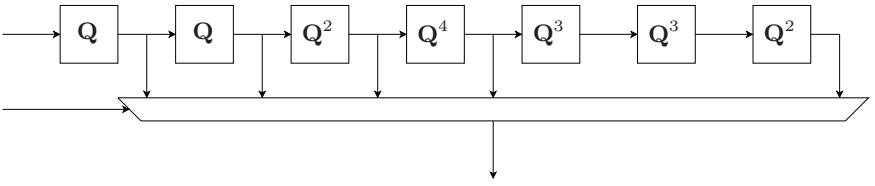

Fig. 3. The repeated squarer of Example 6

The First Solution. Let $E = \{e_1, \ldots, e_\ell\}$, where $e_i > e_{i-1}$ for all i, denote the set of exponents supported by a repeated squarer and let $\Delta_i = e_i - e_{i-1}$ (with $e_0 = 0$). The first solution is to produce fixed exponent circuits for each Δ_i by using the tools of Sec. 4.1 and to concatenate the resulting circuits in increasing order starting from Δ_1. A multiplexer with i as the selector is used to collect the wanted result from the chain.

Example 6. Consider using 6-LUTs (e.g., Virtex-5) for implementing an area optimized repeated squarer for NIST $\mathbb{F}_{2^{233}}$ that supports the exponents: $E = \{1, 2, 4, 8, 16\}$. We get $\Delta_1 = 1$, $\Delta_2 = 1$, $\Delta_3 = 2$, $\Delta_4 = 4$, and $\Delta_5 = 8$. Using exhaustive search for each Δ_i with area minimization as an optimization strategy and concatenating the resulting circuits and a multiplexer gives the circuit depicted in Fig. 3. Area and delay estimates (without the multiplexer) are 1600 LUTs and 8 LUTs, respectively.

Remark 2. The first solution is a generalization of the circuitry presented by Rebeiro and Mukhopadhyay [7]. Their circuitry, called quad-block[4], computes $a^{4^s}(x)$ with $s \in \{2, 3, 4, 5, 7, 9\}$. It is a special case of the first solution which is implemented for an exponent set $E = \{4, 6, 8, 10, 14, 18\}$ with $\mathbf{Q}^2$ blocks only. They showed that the circuit can be efficiently used for accelerating inversions. They used the addition chain $(1, 2, 3, 6, 7, 14, 28, 29, 58, 116, 232)$ for the inversion whereas $(1, 2, 4, 8, 16, 32, 64, 128, 192, 224, 232)$, where e is a power of two in all repeated squarings, would make the circuit simpler, i.e., the same latency could be achieved with $E = \{1, 2, 4, 8, 16\}$ (the circuit of Example 6).

The Second Solution. If a squarer must support all exponents from a given range, then $\Delta_i = 1$ for all i and the first solution results in a squaring chain from which the output is selected with a multiplexer from the outputs of each squarer in the chain. This is clearly very inefficient and we present the following second solution to overcome this problem.

The second solution splits computation for two chains, the first of which is a concatenation $\mathbf{Q}^{e_{\text{opt}}}||\mathbf{Q}^{e_{\text{opt}}}||\ldots||\mathbf{Q}^{e_{\text{opt}}}$ with a length of $\lfloor e_{\max}/e_{\text{opt}} \rfloor$ and the

[4] Squaring and quading are not supported by the quad-block, but they are available elsewhere in the processor.

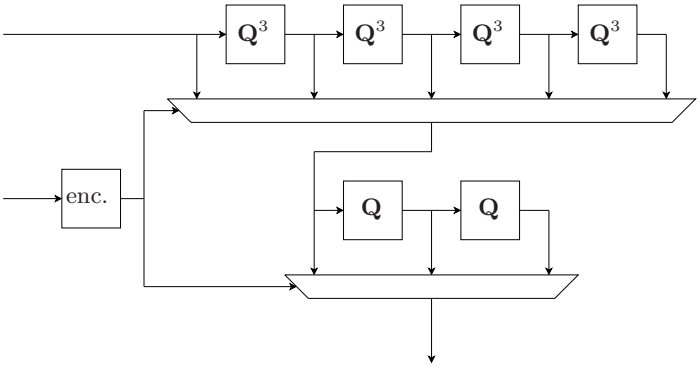

Fig. 4. The repeated squarer of Example 7

second is a square chain $(\mathbf{Q}||\mathbf{Q}||\ldots||\mathbf{Q})$ with a length of $e_{\text{opt}} - 1$. The input to the second chain is selected from the input, the intermediate values, and the output of the first chain with a $(\lfloor e_{\max}/e_{\text{opt}} \rfloor + 1)$-to-1 multiplexer. The output of the entire repeated squarer is obtained with an e_{opt}-to-1 multiplexer from the square chain. The select signals of the multiplexers are derived from e with a simple encoder.

Example 7. Consider using 6-LUTs (e.g. Virtex-5) for implementing a repeated squarer for NIST $\mathbb{F}_{2^{233}}$ that supports exponents in the range $0 \leq e \leq e_{\max} = 14$. Using the tools of Sec. 3 with area optimization, we get $e_{\text{opt}} = 3$. Thus, the first chain consists of four repeated squarers for $\mathbf{Q}^3$ and the second chain is a square chain with two squarers. The repeated squarer is depicted in Fig. 4. Area and delay estimates (without the multiplexers and encoder) are 1238 LUTs and 6 LUTs, respectively.

Remark 3. The second solution results in an exponent range $0 \leq e \leq e_{\text{upper}} = (\lfloor e_{\max}/e_{\text{opt}} \rfloor + 1)\, e_{\text{opt}} - 1$ from which it follows that $e_{\text{upper}} = e_{\max}$ if and only if $e_{\text{opt}} \mid e_{\max} + 1$, else $e_{\max} < e_{\text{upper}} < e_{\max} + e_{\text{opt}}$. If a lower bound starting from one is wanted, it can be easily realized either by designing the encoder so that it does not allow exponent $e = 0$ or by attaching a squarer in front of the repeated squarer. However, the feature that also $a^{2^0}(x) = a(x)$ is supported can be very useful because it allows using the repeated squarer directly as a dummy operation.

Remark 4. Delay can be reduced by replacing the square chain with direct repeated squarers. For instance, in the case of Example 7 (Fig. 4), the second $\mathbf{Q}$ would be replaced by $\mathbf{Q}^2$ taking its input directly from the multiplexer. This would reduce the delay to 5 LUTs but increase area by 77 LUTs.

5 Implementation Results

All VHDL was generated automatically with Matlab scripts[5] (Matlab 7.7.0.471 (R2008b)). The VHDL is device independent. We compiled the code for two Xilinx FPGAs: Spartan-3A 3S1400A-5 ($n = 4$) and Virtex-5 5VLX50-3 ($n = 6$) which represent low-cost and high-end FPGAs, respectively. Synthesis and place&route were performed with Xilinx ISE 10.1.03 WebPACK using default options. The inputs and outputs were registered, otherwise the whole chip area was devoted for the repeated squarers. Several different designs were compiled and Table 2 collects the results. The areas in Table 2 represent the areas of combinatorial parts and the delays give the critical paths from the input registers to the output registers as reported by Xilinx ISE.

5.1 Discussion on the Results

In order to be feasible in practice, the repeated squarers should consume only moderate area and have a delay that is shorter than the critical paths of existing elliptic curve cryptography processors (this ensures that repeated squarings do not become the bottleneck for clock frequency). To the best of our knowledge, the two fastest FPGA-based elliptic curve processors have been presented by Chelton and Benaissa [17] for general binary curves and Järvinen and Skyttä [6] for Koblitz curves. The areas of the processors are 26364 4-LUTs (Virtex-4 4VLX200-11) and 34604 7-LUTs (Adaptive LUTs of Stratix II S180C3), respectively [17,6]. Compared to these, the areas listed in Table 2 are small. The critical paths of the processors are 6.50 ns and 5.33 ns, respectively [17,6]. The delays in Table 2 are of the same magnitude. However, both processors use NIST $\mathbb{F}_{2^{163}}$, whereas we provided results for a larger field, $\mathbb{F}_{2^{233}}$. It is likely that both area and critical paths of the processors would be significantly larger with the larger field. Hence, it is safe to say that the proposed repeated squarers are, indeed, feasible components for elliptic curve cryptography processors.

The delays of the circuits computing $\mathbf{Q}^e$ directly are only slightly faster (and in some cases even slower) than the delays of concatenated circuits. The reasons for this originate from the more difficult place&route of direct circuits that degrades their results. Hence, the proposed repeated squarers can be very competitive against direct circuits even with respect to delay.

Table 2 shows that (with only few exceptions) the setup declared as area optimal by the analysis, indeed, is the best concatenation also after the compilation. Of course, the number of compiled concatenations for each set of exponents is rather small; hence, it is not clear whether a concatenation resulting in an even smaller area exists or not. Most of the exceptions are for Spartan-3A. This is due to the fact that based on the analysis $e_{\text{opt}} = 2$ (see Table 1) whereas in reality based on the values from Table 2 it is $e_{\text{opt}} = 3$. This difference is caused by resource sharing between rows that was not considered by the analysis. The resource sharing between rows is not the only synthesis optimization which is not

[5] The scripts are available at `http://www.tcs.hut.fi/~kjarvine/codes/`

Table 2. Results for $\mathbb{F}_{2^{233}}$ on Spartan-3A 3S1400A-5 and Virtex-5 5VLX50-3

		Spartan-3A		Virtex-5	
e	**Concatenation**[1]	**Area** (LUTs)	**Delay** (ns)	**Area** (LUTs)	**Delay** (ns)
1	1 $_{(4,6)}$	153	3.45	153	2.78
2	2 $_{(4,6)}$	230	4.61	230	3.01
3	3 $_{(4,6)}$	289	5.82	233	2.52
4	2,2 $_{(4)}$	436	5.99	299	2.86
4	4 $_{(6)}$	433	6.09	330	2.89
5	3,2 $_{(4,6)}$	546	7.01	372	3.58
5	5	656	6.52	476	3.63
6	2,2,2 $_{(4)}$	711	6.78	651	3.51
6	3,3 $_{(6)}$	637	6.91	434	4.06
6	6	996	6.59	659	3.55
8	2,2,2,2 $_{(4)}$	939	6.62	1415	4.49
8	3,3,2 $_{(6)}$	916	7.80	587	5.00
8	8	2090	7.27	1576	4.47
16	2,...,2 $_{(4)}$	1691	12.30	1525	8.25
16	4,3,...,3 $_{(6)}$	1733	12.35	1368	8.86
16	16	5727	12.89	4398	6.48
$\{1, 2, 4\}$	1,1,2 $_{(4,6)}$	810	5.77	580	4.07
$\{1, 2, 4, 8\}$	1,1,2,2,2 $_{(4)}$	1350	8.85	1227	5.30
$\{1, 2, 4, 8\}$	1,1,2,4 $_{(6)}$	1396	7.80	1189	5.83
$\{1, 2, 4, 8, 16\}$	1,1,2,...,2 $_{(4)}$	2694	14.82	2015	8.07
$\{1, 2, 4, 8, 16\}$	1,1,2,4,3,3,2 $_{(6)}$	2784	15.68	1823	8.23
$0 \leq e \leq 7$	2 $_{(4)}$	1365	8.40	1319	6.03
$0 \leq e \leq 11$	2 $_{(4)}$	2525	12.65	2113	7.72
$0 \leq e \leq 11$	3 $_{(6)}$	2005	11.00	1809	8.10
$0 \leq e \leq 13$	2 $_{(4)}$	2412	15.78	2694	8.66
$0 \leq e \leq 14$	3 $_{(6)}$	2773	13.58	1911	8.67

[1] The subscript shows n for which the concatenation is area optimal based on the analysis.

considered in the analysis. The synthesis also performs optimizations between different concatenated blocks.

In general, compensating these deficiencies in the analysis without any feedback from the synthesis is extremely difficult because optimizations depend heavily on the target device (on more than simply n) and the synthesis program as well as on the options given for the synthesis. However, the following procedure using feedback from the synthesis compensates the resource sharing between rows of $\mathbf{Q}^e$ nearly completely:

1. Synthesize repeated squaring matrices $\mathbf{Q}^{\hat{e}}$ starting from $\hat{e} = 1$ until e_{opt} is found.
2. Collect the areas from the results of the synthesis and use them as $\mathcal{A}_n(\mathbf{Q}^e)$ in the analysis instead of the estimates given by (6).

This procedure takes resource sharing between rows into account in the analysis and makes the predicted values considerably more accurate. Synthesis programs typically include options that prevent optimizations between design blocks and they can be used for improving the accuracy of the analysis. However, we did not use such options, because the optimizations between blocks typically reduce both area and delay considerably from the predicted values and, hence, preventing them reduces the quality of results.

6 Conclusions

Previously in the paper, we analyzed repeated squarings and presented several possibilities of how to realize repeated squarers. The results presented in Sec. 5 proved the feasibility of repeated squarers by showing that they can be implemented with reasonable area and that they are fast enough not to become the bottleneck.

We conclude by discussing the following benefits of using repeated squares compared to the existing solutions:

Faster Inversions in Binary Fields. Repeated squarers (the first solution) offer a relatively cheap way to implement fast inversions. For example, an Itoh-Tsujii inversion in NIST $\mathbb{F}_{2^{233}}$ requires 10 multiplications and 232 squarings. These squarings can be computed with only 19 repeated squarings using a repeated squarer (first solution) with $E = \{1, 2, 4, 8, 16\}$ (a component that was shown practical in Sec. 5). Speedups up to 88% can be achieved with this repeated squarer compared to an iterative repeated squarer. Naturally, the faster the multiplications are and the larger is the ratio of squarings compared to multiplications, the larger are the speedups. See Appendix A.1 for more information on the computation model, latencies, and speedups with different repeated squarers.

Faster General Exponentiations in Binary Fields. Inversion using Fermat's Little Theorem is simply an exponentiation to the power $2^m - 2$ in the field. Obviously, repeated squarers can be used for accelerating any exponentiation in a similar way and, again, the larger is the ratio of squarings compared to multiplications, the better improvements are achievable.

Faster Scalar Multiplications on Koblitz Curves. With an iterative squarer computing e successive Frobenius maps requires either $2e$ or $3e$ clock cycles depending on the coordinate system. Repeated squarers (the second solution) reduce this to $2\lceil e/e_{\max} \rceil$ or $3\lceil e/e_{\max} \rceil$ clock cycles. Defining speedups in the case of scalar multiplications is not as straightforward as it was for Itoh-Tsujii inversions, because they depend on k that varies. We ran 100000 experiments with

random width-2 τNAFs using two computation models based on existing elliptic curve processors in order to determine the speedups. According to these experiments, repeated squarers can lead to average speedups of over 13% in the latency of scalar multiplication. Expectedly, the faster are the multipliers the larger are the speedups also in this case. Increasing the width ω of τNAF also increases the speedups. Furthermore, methods that reduce either the memory consumption of window methods [18] or the weight $w(k)$ [19] by using more Frobenius maps have been proposed recently and efficient computation of repeated squarings is essential for them. Appendix A.2 presents details on the experiments.

Improved Side-Channel Resistivity. Replacing an iterative repeated squarer with a repeated squarer (the second solution) makes attacking scalar multiplications on Koblitz curves with side-channel attacks considerably harder. Instead of counting clock cycles from the power trace, the adversary must be able to distinguish the exponent e from a single clock cycle in the power trace, i.e, after observing a single clock cycle in the trace, the adversary only knows that the number of Frobenius maps is $\leq e_{\max}$. Therefore, launching a successful side-channel attack is considerably more difficult. If $e_{\max}$ is small and/or ω large, situations where $e > e_{\max}$ occur commonly (see Appendix A.2) which may lead to certain side-channel weaknesses. It may also be possible to learn some information about e from the power consumption of the repeated squarer, e.g., with differential power analysis. Hence, it is clear that this approach does not remove the possibility of a successful power analysis attack entirely, but it is equally clear that it makes attacking significantly harder.

6.1 Future Research

Designing repeated squarers still requires some trial-and-error type optimizations, mainly because resource sharing is hard to incorporate into the analysis. However, the trials are easy and fast to do thanks to the automated VHDL generation provided by the Matlab scripts. Nonetheless, we are searching for heuristics compensating resource sharing.

Modern FPGAs commonly have a structure which cannot be modelled accurately with simple n-LUTs. As a consequence, optimizations for more advanced LUT structures, such as, Stratix ALUTs, 6-to-2 bit LUTs, etc., will be a topic for future research.

Square root can be implemented with fewer resources than squaring if the irreducible is a trinomial [20]. Hence, it might be possible to reduce the complexity of a repeated squaring (especially, varying exponent with the second solution) by, first, "shooting over" the required exponent e with an e_{opt} chain and, then, reversing back with (repeated) square roots, rather than using fewer e_{opt}'s and then reaching e with a few repeated squarings.

The effects of the LUT size, in general, have been surprisingly little studied considering how popular FPGAs are for implementing finite field arithmetic. As shown in this paper, the LUT structure may have significant consequences and could, therefore, open ways to further optimize existing designs on FPGAs;

hence, also other operations, such as finite field multiplications, should be studied from this point of view.

Acknowledgments

This work was supported by the European Commission's 7th Framework Programme (FP7) under contract number ICT-2007-216499 (CACE). The author would like to thank Billy Bob Brumley and the anonymous reviewers for valuable comments and improvement suggestions.

References

1. Miller, V.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
2. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. 48, 203–209 (1987)
3. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Inf. Comput. 78, 171–177 (1988)
4. Guajardo, J., Paar, C.: Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. Designs Codes Cryptogr. 25, 207–216 (2002)
5. Koblitz, N.: CM-curves with good cryptographic properties. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 279–287. Springer, Heidelberg (1992)
6. Järvinen, K., Skyttä, J.: Fast point multiplication on Koblitz curves: Parallelization method and implementations. Microprocess. Microsyst. 33, 106–116 (2009)
7. Rebeiro, C., Mukhopadhyay, D.: High speed compact elliptic curve cryptoprocessor for FPGA platforms. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 376–388. Springer, Heidelberg (2008)
8. Wollinger, T., Guajardo, J., Paar, C.: Security on FPGAs: State-of-the-art implementations and attacks. ACM Trans. Embedd. Comput. Syst. 3, 534–574 (2004)
9. Lutz, J., Hasan, A.: High performance FPGA based elliptic curve cryptographic co-processor. In: International Conference on Information Technology: Coding and Computing, vol. 2, pp. 486–492. IEEE Computer Society, Los Alamitos (2004)
10. National Institute of Standards and Technology (NIST): Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186-2 (2000)
11. Ahmadi, O., Hankerson, D., Rodríguez-Henríquez, F.: Parallel formulations of scalar multiplication on Koblitz curves. J. Univers. Comput. Sci. 14, 481–504 (2008)
12. Gordon, D.M.: A survey of fast exponentiation methods. J. Algorithms 27, 129–146 (1998)
13. Solinas, J.A.: Efficient arithmetic on Koblitz curves. Designs Codes Cryptogr. 19, 195–249 (2000)
14. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in $GF(2^m)$. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 201–212. Springer, Heidelberg (1999)
15. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
16. Hasan, M.A.: Power analysis attacks and algorithmic approaches to their countermeasures for Koblitz curve cryptosystems. IEEE Trans. Comput. 50, 1071–1083 (2001)
17. Chelton, W.N., Benaissa, M.: Fast elliptic curve cryptography on FPGA. IEEE Trans. Very Large Scale Integr (VLSI) Syst. 16, 198–205 (2008)

18. Vuillaume, C., Okeya, K., Takagi, T.: Short-memory scalar multiplication for Koblitz curves. IEEE Trans. Comput. 57, 481–489 (2008)
19. Dimitrov, V.S., Järvinen, K.U., Jacobson, M.J., Chan, W.F., Huang, Z.: Provably sublinear point multiplication on Koblitz curves and its hardware implementation. IEEE Trans. Comput. 57, 1469–1481 (2008)
20. Rodríguez-Henríquez, F., Morales-Luna, G., López, J.: Low-complexity bit-parallel square root computation over $GF(2^m)$ for all trinomials. IEEE Trans. Comput. 57, 471–480 (2008)
21. Al-Daoud, E., Mahmod, R., Rushdan, M., Kilicman, A.: A new addition formula for elliptic curves over $GF(2^n)$. IEEE Trans. Comput. 51, 972–975 (2002)

A Speedup Evaluations

A.1 Itoh-Tsujii Inversion

We consider Itoh-Tsujii inversion in NIST $\mathbb{F}_{2^{233}}$ with the addition chain $\{1, 2, 4, 8, 16, 32, 64, 128, 192, 224, 232\}$ and the following computation model. Let multiplication require M clock cycles and repeated squaring one clock cycle. Assuming that the latency consists of only multiplications and repeated squarings (easily achievable, for example, with the architecture from [6]), an Itoh-Tsujii inversion in NIST $\mathbb{F}_{2^{233}}$ requires $10M + R$ clock cycles where R is the number of repeated squarings. Table 3 lists speedups compared to an iterative squarer.

Table 3. Latencies and speedups of Itoh-Tsujii inversion in $\mathbb{F}_{2^{233}}$ with different repeated squarers and multiplication latencies

E	R	$M = 233$	$M = 18$	$M = 6$	$M = 1$
$\{1\}$	232	2562	412	292	242
$\{1, 2\}$	117	2447(-4.5%)	297(-27.9%)	177(-39.4%)	127(-47.5%)
$\{1, 2, 4\}$	60	2390(-6.7%)	240(-41.7%)	120(-58.9%)	70(-71.1%)
$\{1, 2, 4, 8\}$	32	2362(-7.8%)	212(-48.5%)	92(-68.5%)	42(-82.6%)
$\{1, 2, 4, 8, 16\}$	19	2349(-8.3%)	199(-51.7%)	79(-72.9%)	29(-88.0%)
$\{1, 2, 4, 8, 16, 32\}$	13	2343(-8.5%)	193(-53.2%)	73(-75.0%)	23(-90.5%)
$\{1, 2, 4, 8, 16, 32, 64\}$	11	2341(-8.6%)	191(-53.6%)	71(-75.7%)	21(-91.3%)

A.2 Scalar Multiplication on Koblitz Curves

We consider scalar multiplication on a Koblitz curve NIST K-233 [10] with width-2 τNAF using two computation models. Model 1 represents a generic elliptic curve processors (similar, e.g., to [17]) having one multiplier with a latency M and all other operations having a latency of one. We assume that point additions are computed as proposed in [21] requiring 8 multiplications, 5 squarings, and 8 additions on K-233. Thus, we assume a latency of $(w(k) - 1)(8M + 13) + 3R$ for scalar multiplication (without the final inversion), where R is the number of

repeated squarings per coordinate required in computation of Frobenius maps. Model 2 is taken directly from [6] representing the fastest elliptic curve processor available in the literature. With that processor the latency of scalar multiplication is $(w(k) - 1)(2M + 2) + R$. Table 4 presents results after evaluating both models with 100000 random width-2 τNAFs (obtained as proposed in [13]).

Table 4. Average latencies and speedups in scalar multiplications on NIST K-233 and width-2 τNAF with different repeated squarers and multiplication latencies using two computation models. Coverage gives the percentage of Frobenius maps where one coordinate can be mapped with a single repeated squaring; the value in parentheses gives the percentage of scalars where all Frobenius maps had this property.

$e_{\max}$	Model	Coverage	$M = 17$	$M = 12$	$M = 8$	$M = 5$
1	1	0.33% (0.00%)	12133	9062	6604	4762
2	1	50.40% (0.00%)	11826(-2.5%)	8754(-3.4%)	6297(-4.7%)	4454 (-6.5%)
3	1	75.31% (0.00%)	11738(-3.3%)	8667(-4.4%)	6210(-6.0%)	4367 (-8.3%)
7	1	98.48%(29.57%)	11676(-3.8%)	8605(-5.0%)	6148(-6.9%)	4305 (-9.6%)
11	1	99.91%(93.07%)	11673(-3.8%)	8602(-5.1%)	6145(-7.0%)	4302 (-9.7%)
14	1	99.99%(99.15%)	11673(-3.8%)	8601(-5.1%)	6144(-7.0%)	4301 (-9.7%)
1	2	0.33% (0.00%)	2995	2227	1613	1152
2	2	50.40% (0.00%)	2893(-3.4%)	2125(-4.6%)	1510(-6.4%)	1050 (-8.9%)
3	2	75.31% (0.00%)	2863(-4.4%)	2095(-5.9%)	1481(-8.2%)	1020(-11.4%)
7	2	98.48%(29.57%)	2843(-5.1%)	2075(-6.8%)	1461(-9.4%)	1000(-13.2%)
11	2	99.91%(93.07%)	2842(-5.1%)	2074(-6.9%)	1459(-9.5%)	999(-13.3%)
14	2	99.99%(99.15%)	2842(-5.1%)	2074(-6.9%)	1459(-9.5%)	999(-13.3%)