Efficient Double Bases for Scalar Multiplication

Nicolas Méloni, M. A. Hasan

Efficient Double Bases for Scalar Multiplication

Nicolas Méloni and M. Anwar Hasan

Abstract

In this paper we present efficient algorithms to take advantage of the double-base number system in the context of elliptic curve scalar multiplication. We propose a generalized version of Yao's exponentiation algorithm allowing the use of general double-base expansions instead of the popular double base chains. We introduce a class of constrained double base expansions and prove that the average density of non-zero terms in such expansions is $O\left(\frac{\log k}{\log\log k}\right)$ for any large integer k. We also propose an efficient algorithm for computing constrained expansions and finally provide a comprehensive comparison to double-base chain expansions, including a large variety of curve shapes and various key sizes.

Keywords: Double-base number system, elliptic curve, point scalar multiplication, Yao's algorithm.

I. Introduction

Since its introduction to modular exponentiation by Dimitrov et. al. [1], double-base (DB) expansions have shown to be a very attractive alternative to the classical non-adjacent form (NAF), window NAF and fractional window NAF. An integer k is written as a sum of 2-3 integers, i.e. numbers of the form 2^b3^t. Such expansions are really redundant and it can be proven that, among all of them, very sparse expansions (with $O\left(\frac{\log k}{\log\log k}\right)$ terms) can be effectively computed via a greedy algorithm.

Despite those advantages, this number system has two drawbacks. First, computing DB expansions using the greedy algorithm is quite slower than any recoding algorithm used to compute window NAF expansions. Second, they seem to be relevant only in the context of exponentiation of a fixed element. The first issue was partially solved by Berthé and Imbert [2]. They proposed an asymptotically faster algorithm to compute DB expansions, which for cryptographically relevant key sizes, provides a speed up of 40%. Another approach to that issue was proposed by Doche and Imbert [3]. It consists of precomputing the binary representations of 3^t up to some bound, sort them in lexicographic order and look for the 2-3 integer closest to some number by dichotomic search. The second issue was overcome by the introduction of double base chain (DBC) expansions where the integer $k = \sum 2^{b_i}3^{t_i}$ is still represented as a sum of 2-3 integers but with the restriction that (b_i) and (t_i) must be two decreasing sequences [4]. The restriction causes the number of terms to increase, but makes it possible to perform the scalar multiplication using a Horner like scheme.

In this work we take a different approach to address both of the aforementioned issues of general DB expansions. In Section III, we propose a generalized version of Yao's exponentiation algorithm adapted to double bases. By imposing a maximum bound on b_i's and t_i's that is clearly less restrictive than the DBC condition, we show that our expansion method provides significant improvement even when compared to the optimized DB methods. We provide a complexity analysis of our constrained double-base (CDB) expansions and prove that the number of terms of those expansions still is in $O\left(\frac{\log k}{\log\log k}\right)$, for k large enough. In Section V we tackle the problem of efficient computation of expansions and introduce a window greedy algorithm. We also propose a method to compute sparser expansions than the basic greedy algorithm. Finally, in Section VI, we perform a comprehensive study of the performance of our algorithms. In particular we show that the window greedy algorithm provides a factor 10 speed-up compared to the classical greedy algorithm. Also, we provide comparisons between CDB expansions and DBC on a wide range of elliptic curve forms and for various key sizes.

II. Background

In this section, we give a brief review of the background material used in the paper.

A. Elliptic curves

Definition 2.1: An elliptic curve E over a field K denoted by E/K is given by the equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ are such that, for each point (x, y) on E, the partial derivatives do not vanish simultaneously.

In this paper, we only deal with curves defined over a prime finite field ($K = \mathbb{F}_p$) of characteristic greater than 3. In this case, the curve equation can be simplified to

$$y^2 = x^3 + ax + b$$

where $a, b \in K$ and $4a^3 + 27b^2 \neq 0$. The set of points $E(K)$ defined over K forms an abelian group. Elliptic curve group law computation has been a very active research area over the past

years. Many formulas and coordinate systems have been proposed for which one can refer to [5], [6] for a comprehensive overview.

Another feature of the elliptic curve group law is that it allows fast composite operations as well as different types of additions. To take full advantage of our point scalar multiplication method, and in addition to the classical addition (**ADD**) and doubling (**DBL**) operations, we consider the following operations:

- **tripling (TPL)**: point tripling
- **readdition (reADD)**: addition of a point that has been previously added to another point
- **mixed addition (mADD)**: addition of a point in affine coordinate (i.e., $Z = 1$) to another point

In addition to various coordinate systems and composite operations, many curve shapes have been proposed to improve group operation formulas. In this paper, we will consider a variety of curve shapes including:

- tripling oriented Doche-Icart-Kohel curves (3DIK) [7]
- Edwards curves (Edwards) [8], [9] with inverted coordinates [10]
- Hessian curves [11], [12], [13]
- Extended Jacobi Quartics (ExtJQuartic) [11], [14], [12]
- Jacobi intersections (JacIntersect) [11], [15]
- Jacobian coordinates (Jacobian) with the special case $a_4 = -3$ (Jacobian-3).

Table I summarizes the cost of those operations on all the considered curves, where M and S correspond to field multiplication and squaring, respectively. Finally, some more optimizations can be found in [16], [17] for the quintupling formulas. One can also refer to [18] for an extensive overview of different formulas, coordinates systems, curve shapes and their latest updates.

B. Double-base number system

Let k be an integer. As mentioned earlier, one can represent k as $\sum_{i=1}^{n} 2^{b_i} 3^{t_i}$. Such a representation always exists. In fact, this number system is quite redundant. One of the most interesting properties is that, among all the possible representations for a given integer, some of them are really sparse, that is to say that the number of non-zero terms is quite low.

To compute such expansions, one typically uses Algorithm 1. It consists of the following: find

Curve shape	DBL	TPL	ADD	reADD	mADD
3DIK	2M+7S	6M+6S	11M+6S	10M+6S	7M+4S
Edwards	3M+4S	9M+4S	10M+1S	10M+1S	9M+1S
ExtJQuartic	2M+5S	8M+4S	7M+4S	7M+3S	6M+3S
Hessian	3M+6S	8M+6S	6M+6S	6M+6S	5M+6S
InvEdwards	3M+4S	9M+4S	9M+1S	9M+1S	8M+1S
JacIntersect	2M+5S	6M+10S	11M+1S	11M+1S	10M+1S
Jacobian	1M+8S	5M+10S	11M+5S	10M+4S	7M+4S
Jacobian-3	3M+5S	7M+7S	11M+5S	10M+4S	7M+4S

TABLE I
ELLIPTIC CURVE OPERATIONS COST.

Algorithm 1 Greedy algorithm for computing DB expansion

Input: $k \in \mathbb{N}$
Output: $((b_i, t_i))_i$ such that $k = \sum_{i=1}^{n} 2^{b_i} 3^{t_i}$

1: $i \leftarrow 0$
2: **while** $k > 0$ **do**
3: Computes $c = 2^b 3^t$ the largest 2-3 integer smaller than k
4: $b_i \leftarrow b$, $t_i \leftarrow t$, $k \leftarrow k - c$, $i \leftarrow i + 1$
5: **end while**
6: **return** $((b_i, t_i))_i$

the largest integer of the form $2^{b_i} 3^{t_i}$ smaller than k, subtract it from k and repeat this process with $k \leftarrow k - 2^{b_i} 3^{t_i}$ until k is equal to zero.

The basic method to perform a point multiplication using a DB expansion is to compute the points $[2^{b_i} 3^{t_i}]P$ for $i = 1$ to n and add them all. In practice, this method does not provide an efficient way to perform a point multiplication as the low number of additions does not compensate the many doublings and triplings. That is why the *general* DB representation has been considered to be not that advantageous for point scalar multiplication.

To overcome this problem, Dimitrov, Imbert, and Mishra [4] have introduced the concept of DBC. In this system, k is still represented as $\sum_{i=1}^{n} 2^{b_i} 3^{t_i}$, but with the restriction that (b_i) and (t_i) must be two decreasing sequences, allowing a Horner-like evaluation of kP using only b_1 doublings and t_1 triplings. The main drawback of this method is that it significantly increases the number of point additions.

A number of improvements have been proposed by applying various modifications including the possibility to use digits from a larger set than $\{0, 1\}$ [3], the use of multiple bases [16], etc. One can refer to [19] for an overview of the latest optimizations.

III. YAO'S ALGORITHM FOR DOUBLE BASES

A. Original Yao's algorithm

Published in 1976 [20], Yao's algorithm can be seen as the right-to-left counterpart of Brauer's algorithm. Let $k = k_{l-1}2^{l-1} + \cdots + k_1 2 + k_0$ with $k_i \in \{0, 1, \ldots, 2^w - 1\}$, for some w. The algorithm first computes $2^i P$ for all i lower than l by successive doublings. Then it computes $d(1)P, \ldots, d(2^w - 1)P$, where $d(j)$ is the sum of the 2^i's such that $k_i = j$. Said differently, it mainly consists of considering the integer k as

$$1 \times \underbrace{\sum_{k_i=1} 2^i}_{d(1)} + 2 \times \underbrace{\sum_{k_i=2} 2^i}_{d(2)} + \cdots + (2^w - 1) \times \underbrace{\sum_{k_i=2^w-1} 2^i}_{d(2^w-1)}.$$

We can see that $d(1)$ is the sum of all the powers of 2 associated to digit 1, $d(2)$ is the sum of all the powers of 2 associated to digit 2, etc. Finally kP is obtained as $d(1)P + 2d(2)P + \cdots + (2^w - 1)d(2^w - 1)P$. In order to save some group operations, it is usually computed as $d(2^w - 1)P + (d(2^w - 1)P + d(2^w - 2)P) + \cdots + (d(2^w - 1)P + \cdots + d(1)P)$.

Example 3.1: Let $k = 314159$. We have $\mathrm{NAF}_3(k) = 100\,0300\,1003\,0000\,5007$, $l = 19$ and $2^w - 1 = 7$. One can compute kP in the following way:

- consider k as $1 \times (2^{18} + 2^{11}) + 3 \times (2^{14} + 2^8) + 5 \times 2^3 + 7 \times 2^0$
- compute $P, 2P, 4P, \ldots, 2^{18}P$
- $d(1)P = 2^{18}P + 2^{11}P$, $d(3)P = 2^{14}P + 2^8 P$, $d(5)P = 2^3 P$, $d(7)P = P$
- $kP = 2(d(7)P) + 2(d(7)P + d(5)P) + 2(d(7)P + d(5)P + d(3)P) + d(7)P + d(5)P + d(3)P + d(1)P = 7d(7)P + 5d(5)P + 3d(3)P + d(1)P$

In this example, we have:

$$\begin{aligned} d(1) &= 100\,0000\,1000\,0000\,0000 \\ d(3) &= 000\,0100\,0001\,0000\,0000 \\ d(5) &= 000\,0000\,0000\,0000\,1000 \\ d(7) &= 000\,0000\,0000\,0000\,0001 \\ k &= 100\,0300\,1003\,0000\,5007 \\ &= 7d(7) + 5d(5) + 3d(3) + d(1) \end{aligned}$$

B. Generalized Yao's algorithm

Let $\mathcal{A} = \{a_1, \ldots, a_r\}$ and $\mathcal{B} = \{b_1, \ldots, b_t\}$ be two sets of integers. Let k be an integer that can be written as $\sum_{i=1}^{n} a_{f(i)} b_{g(i)}$ with $f : \{1, \ldots, n\} \to \{1, \ldots, r\}$ and $g : \{1, \ldots, n\} \to \{1, \ldots, t\}$. It is possible to use a generalized version of Yao's algorithm to compute kP. To do so, we first compute $b_i P$'s, for $i = 1 \ldots t$. Then, for $j = 1 \ldots r$, we compute $d(j)P$ as the sum of all $b_{g(i)}P$'s such that $f(i) = j$. In other words, $d(1)P$ will be the sum of all $b_{g(i)}P$'s associated to a_1, $d(2)P$ will be the sum of all $b_{g(i)}P$'s associated to a_2, etc. Finally, $kP = a_1 d(1)P + a_2 d(2)P + \cdots + a_n d(n)P$.

It is easy to see that with a proper choice of $\mathcal{A}$ and $\mathcal{B}$, we find again forms of the Yao algorithm that are of our interest. For instance, the original version of Yao's algorithm is associated to the sets $\mathcal{A} = \{1, 2, \ldots, 2^n\}$ and $\mathcal{B} = \{1, 3, 5, \ldots, 2^w - 1\}$ and the double-base version to $\mathcal{A} = \{1, 2, \ldots, 2^{b_{max}}\}$ and $\mathcal{B} = \{1, 3, \ldots, 3^{t_{max}}\}$. Using the latter form, one can perform a scalar multiplication with Algorithm 2.

Algorithm 2 Double-base version of Yao's scalar multiplication algorithm

Input: G a group, $P \in G$ and $k \geq 0$

Output: $Q = kP$

1: Compute a DB expansion for $k = \sum_{i=1}^{n} 2^{b_i} 3^{t_i}$
2: **for** $0 \leq i \leq \max(t_j)$ **do**
3: $\quad P_i \leftarrow 3^i P$
4: **end for**
5: **for** $1 \leq i \leq n$ **do**
6: $\quad Q_{b_i} \leftarrow Q_{b_i} + P_{t_i}$
7: **end for**
8: $Q \leftarrow Q_{\max(b_j)}$
9: **for** $i = \max(b_j) - 1$ **down to** 0 **do**
10: $\quad Q \leftarrow 2Q + Q_i$
11: **end for**
12: **return** Q

Example 3.2: Let $k = 281409$. One of its DB expansions is: $k = 2^7 3^7 + 2^4 3^4 + 2^2 3^3 + 2^1 3^2 + 2^4 3^1 + 2^0 3^1$, so that $(b_1, \ldots, b_6) = (7, 4, 2, 1, 4, 0)$ and $(t_1, \ldots, t_6) = (7, 4, 3, 2, 1, 1)$. With Algorithm 2, one can compute kP in the following way:

- compute $P_0 = P, P_1 = 3P, P_2 = 3^2 P, \ldots, P_7 = 3^7 P$
- $Q_{b_1} = Q_7 \leftarrow P_{t_1} (= P_7)$
- $Q_{b_2} = Q_4 \leftarrow P_{t_2} (= P_4)$
- $Q_{b_3} = Q_2 \leftarrow P_{t_3} (= P_3)$
- $Q_{b_4} = Q_1 \leftarrow P_{t_4} (= P_2)$
- $Q_{b_5} = Q_4 \leftarrow Q_4 + P_{t_5} (= P_1 + P_4)$
- $Q_{b_6} = Q_0 \leftarrow P_{t_6} (= P_1)$.

Finally, the last **for** loop is a Horner scheme to compute $kP = 2(2^2(2^3 Q_7 + Q_4) + Q_2) + Q_1) + Q_0 = (2^7 3^7 + 2^4(3^4 + 3^1) + 2^2 3^3 + 2^1 3^2 + 2^0 3^1)P$.

IV. Constrained double-base representation

The DB version of Yao's algorithm is optimal in the sense that it requires $\max(b_i)$ doublings, $\max(t_i)$ triplings and $n - 1$ additions. However, the numbers of doublings and triplings are

independent, which means that $2^{\max(b_i)}3^{\max(t_i)}$ can be quite larger than k. For instance, with parameter $k = 2219$, the greedy algorithm returns $3^7 + 2^5$ so that $2^{\max(b_i)}3^{\max(t_i)} = 3^7 2^5 = 69984$. This means that Yao's algorithm would perform enough doublings and triplings to compute a 17-bit integer. Despite the sparseness of the representation (i.e. the small value of n), most of the time it will end up in slowing down the whole process. In order to reduce the computation time, we propose a slightly more constrained version of the DB number system by setting maximum bounds b_{max} and t_{max} for both the b_i's and the t_i's so that $2^{b_{max}}3^{t_{max}} \sim k$. It can be seen as an intermediate representation between the general DB, where no constraint applies, and DBC, where the boundaries get lower as the DB recoding goes.

In that context, it becomes unclear if Theorem 4 from [21] (refer to as Dimitrov's theorem in the rest of the paper) still holds, that is to say whether or not the number of terms of those expansions is sub-linear in the size of k, like DBC expansions. Theorem 4.1 shows that, under reasonable hypothesis, CDB expansions stay sub-linear.

Theorem 4.1: Let c_1, c_2 be two positive real numbers such that $c_1 + c_2 \geq 1$. Then, for k large enough, the greedy algorithm with parameter k and bounds $b_{max} = \lfloor c_1 \log_2(k) \rfloor + 1$ and $t_{max} = \lfloor c_2 \log_3(k) \rfloor + 1$ terminates in $O\left(\frac{\log k}{\log\log k}\right)$ steps.

Proof: Let us define $T_{2,3}(k) = \{2^b 3^t \leq k\}$ and $\bar{T}_{2,3}(k) = \{2^b 3^t \leq k : b \leq b_{max} \text{ and } t \leq t_{max}\}$. Without loss of generality, we suppose that $3^{t_{max}} < 2^{b_{max}}$. We split the proof into three cases: $k \leq 3^{t_{max}}$, $3^{t_{max}} \leq k \leq 2^{b_{max}}$ and $2^{b_{max}} \leq k$.

Case 1: $k \leq 3^{t_{max}}$.

We remark that if $k \leq \min(2^{b_{max}}, 3^{t_{max}})$, then $T_{2,3}(k) = \bar{T}_{2,3}(k)$ so that the greedy and the constrained greedy algorithms return the same results. Thus, from Dimitrov's theorem, our theorem holds.

Case 2: $3^{t_{max}} \leq k \leq 2^{b_{max}}$.

Let B be the smallest integer such that $\frac{3^{t_{max}}}{2} \leq \frac{k}{2^B} \leq 3^{t_{max}}$. For k large enough, Tijdeman's theorem [22] applied to $K = \frac{k}{2^B}$ guarantees that there exists $2^b 3^t \in T_{2,3}(K)$ such that:

$$K - 2^b 3^t \leq \frac{K}{(\log K)^C},$$

for some absolute constant C. Obviously $2^{b+B}3^t$ belongs to $\bar{T}_{2,3}(k)$ and satisfies

$$k - 2^{b+B}3^t \leq \frac{k}{(\log K)^C} \leq \frac{k}{(t_{max}\log(3) - \log(2))^C} \leq \frac{k}{(c_2' \log k)^C}.$$

In other words, there is always a number from $\bar{T}_{2,3}(k)$ larger than $k - \frac{k}{(c_2' \log k)^C}$. Obviously, the largest integer from $\bar{T}_{2,3}(k)$ satisfies the previous propriety and from Dimitrov's theorem, we conclude that the constrained greedy algorithm terminates in $O(\frac{\log k}{\log\log k})$ steps.

Case 3: $2^{b_{max}} \leq k$.

Let $k_0 = k$. From the constrained greedy algorithm we construct a sequence $k_0 > k_1 > \cdots > k_l$ such that $k_{i+1} = k_i - 2^{b_{i+1}}3^{t_{i+1}}$. By definition of b_{max} and t_{max} we know that $k/2 \leq 2^{b_{max}}3^{t_{max}}$, thus there exists an integer $d = 2^B 3^T$ from $\bar{T}_{2,3}(k)$ such that $k/2 \leq d \leq k$. More generally, it ensures that the sequence (k_i) satisfies $k_{i+1} \leq k_i/2$, hence $k_i \leq k/2^i$. Let $n = \left\lfloor \frac{\log k}{\log\log k} \right\rfloor$. We suppose k is large enough such that the Tijdeman theorem applies to any integer larger than $2^n/24$. To proceed we need the following result:

Lemma 4.2: For any integer k' smaller than k_n there exist $d \in \bar{T}_{2,3}(k')$ and a constant A such that

$$k' - d \leq \frac{k'}{An^C}.$$

Proof: Let $k' \leq k_n$. Since $k_n \leq \frac{k}{2^n}$, k' satisfies $\frac{k}{k_n} \geq 2^n$. For $0 \leq b \leq B$ and $0 \leq t \leq T$, we define

$$K(b,t) = \frac{k'}{2^{B-b}3^{T-t}} \text{ and } E_{B,T} = \{K(b,t) \leq \min(2^b, 3^t)\}.$$

Clearly, $K(0,0) \in E_{B,T}$ and hence $E_{B,T}$ is not empty. Let $K(\bar{b},\bar{t})$ be its largest element. Then $K(\bar{b}+1,\bar{t}) \notin E_{B,T}$ which means that $K(\bar{b}+1,\bar{t}) > \min(2^{\bar{b}+1}, 3^{\bar{t}})$. We remark that $2^{\bar{b}+1}$ must be larger than $3^{\bar{t}}$, otherwise we would have $K(\bar{b}+1,\bar{t}) > 2^{\bar{b}+1}$ and thus $K(\bar{b},\bar{t}) > 2^{\bar{b}}$, which is absurd. Hence we have $K(\bar{b}+1,\bar{t}) > 3^{\bar{t}}$ which means that

$$\frac{k'}{2^{B-\bar{b}-1}3^{T-\bar{t}}} > 3^{\bar{t}} \Rightarrow 2^{\bar{b}+1} > \frac{2^B 3^T}{k'} \geq \frac{k}{2k'}.$$

The same reasoning applied to $K(\bar{b},\bar{t}+1)$ leads to $3^{\bar{t}+1} > \frac{k}{2k'}$. Now,

$$K(\bar{b},\bar{t}) = \frac{k'}{2^{B-\bar{b}}3^{T-\bar{t}}} \geq \frac{k'}{k} \times \frac{k}{4k'} \times \frac{k}{6k'} \geq \frac{2^n}{24},$$

thus we can apply the Tijdeman result. We get $b' \leq \bar{b}$ and $t' \leq \bar{t}$ such that $K(\bar{b},\bar{t}) - 2^{b'}3^{t'} \leq \frac{K(\bar{b},\bar{t})}{(\log K(\bar{b},\bar{t}))^C}$. Multiplying by $2^{B-\bar{b}}3^{T-\bar{t}}$ we obtain that

$$k' - 2^{B+b'-\bar{b}}3^{T+t'-\bar{t}} \leq \frac{k'}{(\log(K(\bar{b},\bar{t})))^C} \leq \frac{k'}{(n\log(2)-\log(24))^C} \leq \frac{k'}{An^C}.$$

■

Lemma 4.2 guaranties that the constrained greedy algorithm produces a sequence $k_n > k_{n+1} > k_{n+m}$ such that $k_{n+i} - 2^{b_{n+i+1}}3^{t_{n+i+1}} \leq \frac{k_{n+i}}{An^C}$. Then we obtain that

$$k_{n+m+1} \leq \frac{k_n}{A^m n^{C\times m}} \leq \frac{k}{2^n \times A^m n^{C\times m}}.$$

To finish the proof, we show that there exists a function $f : k \mapsto f(k)$ such that we can choose an $m > f(k)$ with $k_{n+m+1} \leq 2^{b_{max}}$ and $f(k)$ not asymptotically bigger than $O\left(\frac{\log k}{\log\log k}\right)$. We have

$$\begin{aligned}\log k_{n+m+1} &\leq \log\left(\frac{k}{2^n \times A^m n^{Cm}}\right)\\ &\leq \log k - n\log 2 - m\log(A) - Cm\log(n)\end{aligned}$$

This is smaller than $b_{max}\log 2 \geq c_1 \log k$ provided that

$$(1-c_1)\log k - \left\lfloor\frac{\log k}{\log\log k}\right\rfloor \log(2) < m\left(\log(A) + C\log\left(\left\lfloor\frac{\log k}{\log\log k}\right\rfloor\right)\right)$$

or, for k large enough and some C'

$$C'\frac{\log k}{\log\log k - \log\log\log k} < m.$$

So, for some constant D larger that C' we set

$$f(k) = D\frac{\log k}{\log\log k - \log\log\log k}.$$

Finally, for k large enough, we have $f(k) = O\left(\frac{\log k}{\log\log k}\right)$, which concludes the proof.

■

V. Efficient double-base expansions

The question of efficiency when dealing with double base expansions in the context of elliptic curve point multiplication is two-fold: the speed at which one can compute the expansions and how good they are for point multiplication. In this section we address both issues by proposing a sliding-window version of the general greedy algorithm 1. The main idea is to take advantage of the boundaries set for b_i's and t_i's in order to reduce the size of the operands in Algorithm 1.

A. The window greedy algorithm

Let k be an n bit integer such that $k > 3^{t_{max}}$ and let 2^b3^t be the 2-3 integer closest to k satisfying $2^b3^t < k$, $b \leq b_{max}$ and $t \leq t_{max}$. Finally, let u be the largest integer such that $k/2^u > 3^{t_{max}}$. From these hypothesis we easily deduce that $b \geq u$. Hence, if $2^{b'}3^{t'}$ is the largest 2-3 integer smaller than $k' = \lfloor k/2^u \rfloor$ satisfying $t \leq t_{max}$ then $2^{b'+u}3^{t'}$ is the largest 2-3 integer smaller than k satisfying $t' \leq t_{max}$ and thus $b = b' + u$ and $t = t'$. This result means that looking for the 2-3 integer closest to k' is sufficient to find the 2-3 integer to closest k. In other words, one can simply consider the highest bits of k to find the closest 2-3 integer and multiply the result by the appropriate power of 2. One can finally compute the full CDB expansion with Algorithm 3. As long as the window size w satisfies $2^{w-1} > 3^{t_{max}}$, the algorithm returns the same expansion as the classical greedy algorithm, i.e. Algorithm 1.

Algorithm 3 Window Greedy algorithm for computing CDB expansion

Input: $k \in \mathbb{N}$ and three parameters b_{max}, t_{max} and w

Output: $((b_i, t_i))_i$ such that $k = \sum_{i=1}^{n} 2^{b_i} 3^{t_i}$ and $\forall i, b_i \leq b_{max}$ and $t_i \leq t_{max}$

$i \leftarrow 0$
while $k > 0$ **do**
    $s \leftarrow$ bit-size(k)
    $u = \max(0, s - w)$
    $k' = k/2^u$
    if $s \leq b_{max}$ **then**
        $b_m \leftarrow w$
    else if $u < b_{max}$ **then**
        $b_m \leftarrow b_{max} - u$
    else
        $b_m \leftarrow 0$
    end if
    Compute $c = 2^{b'} 3^{t'}$ the largest 2-3 integer smaller than k' with $b' \leq b_m$ and $t' \leq t_{max}$
    $b_i \leftarrow b'$, $t_i \leftarrow t'$,
    $c \leftarrow c \times 2^u$, $b_i \leftarrow b_i + u$
    $k \leftarrow k - c$, $i \leftarrow i + 1$
end while
return $((b_i, t_i))_i$

If k' fits in a processor register, we can expect significant speed up by designing a specific function to find the 2-3 integer closest to k'. This is mainly because of the following reasons. First, manipulating smaller operands naturally speeds up arithmetic computations. Second, the data structure management for large integers can be really expensive compared to the cost of the actual arithmetic, especially for small integers. Typically, in our implementations, basic operations such as additions, multiplications or shifts on raw C types (*long long unsigned*) performs six times faster than on 64-bit *gmp* integer type (*mpz_t*).

From Table VI of Section VI-C we can see that optimal choices for t_{max} are usually below 41 which means that k' can be taken smaller than $3^{41} < 2^{64}$, i.e. k' can fit in the registers of almost

any of today's general purpose processors. We have implemented the window greedy algorithm, i.e. Algorithm 3 and compared it to the basic algorithm. We used the `gmp` library to operate large integers and raw C implementation for register size integers. Tables III, IV and V show that using our window method, the greedy algorithm performs about 15 times faster.

B. Shorter double-base expansions

The efficiency of Yao's algorithm is directly linked to the number of terms of the DB expansions of the scalar k. Thus, it is quite natural to look for shorter expansions to improve the overall speed of the algorithm. One obvious way to do so is to allow signed expansions, i.e., scalar k is represented as $\sum s_i 2^{b_i} 3^{t_i}$ where $s_i \in \{-1, 1\}$. In order to obtain such expansions, one just has to look for the 2-3 integer to closest k (instead of the largest one smaller than k) and repeat the process with $|k - 2^b 3^t|$.

An alternative way to compute smaller DB expansions is to look for the integer of the form $s2^b3^t + s'2^{b'}3^{t'}$ closest to k (with $s, s' \in \{-1, 0, 1\}$) and perform the greedy algorithm with this new function. If both s and s' are positive, it is not necessary to try all pairs of the form $(2^b3^t, 2^{b'}3^{t'})$. Indeed, let 2^b3^t be the largest of the two numbers, then it must satisfy $|k - 2^b3^t| < k/2$ (otherwise $2^{b+1}3^t$ or $2^{b+1}3^{t-1}$ would be closer to k than $2^b3^t + 2^{b'}3^{t'}$). Then, for each such 2-3 integer it suffices to look for the 2-3 integer closest to $|k - 2^b3^t|$ and return the best pair. We use the same approach to find the closest signed sum even though, in this case, there is no way to prove a similar result. Indeed, the closest signed sum to k might be made of two 2-3 integers much larger than k. On the other hand, the obtained expansions are still guaranteed to be shorter than their non-signed counterparts.

In practice, one can find a signed sum of two 2-3 integers close enough to k using Algorithm 4.

Algorithm 4 Finding a sum of two 2-3 integers close to k

Input: $k \in \mathbb{N}$,

Output: $c = 2^{b_1}3^{t_1} \pm 2^{b_2}3^{t_2}$ close to k

1: $b \leftarrow \lceil \log_2 k \rceil$, $t \leftarrow 0$, $c \leftarrow 0$
2: $c_1 \leftarrow 2^b 3^t$
3: **while** $b \geq 0$ **do**
4: **if** $k = c_1$ **then**
5: **return** c_1
6: **end if**
7: Compute $c_2 \leftarrow 2^{b'}3^{t'}$ the closest 2-3 integer to $|k - c_1|$
8: **if** $c_1 \leq k$ **then**
9: $c_3 \leftarrow c_1 + c_2$
10: **else**
11: $c_3 \leftarrow c_1 - c_2$
12: **end if**
13: **if** $|k - c_3| < |k - c|$ **then**
14: $c \leftarrow c_3$
15: **end if**
16: **if** $c_1 > k$ **then**
17: $b \leftarrow b - 1$, $c_1 \leftarrow c_1/2$
18: **else**
19: $t \leftarrow t + 1$, $c_1 \leftarrow c_1 \times 3$
20: **end if**
21: **end while**
22: **return** c

Algorithm 4 can easily be generalized to larger combination of 2-3 integers. Such algorithms can be defined recursively by replacing line 7 of Algorithm 4 by the appropriate computation. In any case, if we consider that one can compute the 2-3 integer closest to a given integer k in $O(\log k)$ steps, then finding the sum of t such 2-3 integers requires $O((\log k)^t)$ steps. Such methods become quickly too costly as t grows, that is why we only consider the values $1, 2$ and

3 for t. The resulting window greedy algorithms will be refereed to as, respectively, depth 1, 2 and 3 greedy algorithms or w-greedy(1), (2) and (3).

Remark 5.1: One can obviously apply the same window approach to Berthé and Imbert's algorithm [2] and expect the same kind of improvements. However, in the case of Doche and Imbert's algorithm [3], adapting our approach is not that easy. Let us recall that it consists of precomputing a dictionary with the binary representations of 3^t up to some bound (typically $t = 41$ here), sort them in lexicographic order and look for the closest 2-3 integer to some number by dichotomic search. When looking for the sum of two 2-3 integers closest to some integer k, one can either use the approach of Algorithm 4 to find the 2-3 integer closest to $k' \leftarrow |k - 2^b 3^t|$, for some b's and t's or consider a generalized version of the dictionary approach and precompute all binary representations of $3^t \pm 2^b 3^{t'}$ with some proper bounds.

In the first case, the gain corresponds to the speed ratio between the standard and the dictionary methods. In our implementation, the Doche and Imbert method is approximately 5 times faster and so are the different window versions. In the second case, the dictionary size grows too quickly with the window size to remain practical.

VI. Implementation results

A. Caching strategies

Caching intermediate results while performing an elliptic curve group operation is a very important optimization technique. In this subsection, we show that the use of our generalized algorithm allows some savings that cannot be achieved with the traditional methods. To better clarify this point, we detail our caching strategy for curves in the Weierstrass form using Jacobian coordinates with parameter $a = 3$ (Jac-3). Similar methods are applicable to all different curve types considered.

Addition:

$P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2)$ and $P + Q = (X_3, Y_3, Z_3)$

$$A = X_1 Z_2^2, \quad B = X_2 Z_1^2, \quad C = Y_1 Z_2^3, \quad D = Y_2 Z_1^3, \quad E = B - A,$$
$$F = 2(D - C), \quad G = (2E)^2, \quad H = E \times G, \quad I = A \times G,$$

and

$$X_3 = F^2 - H - 2I \quad Y_3 = F(I - X_3) - 2CH, \quad Z_3 = ((Z_1 + Z_2)^2 - Z_1^2 - Z_2^2)E$$

Doubling:

$2P = (X_3, Y_3, Z_3)$

$$A = X_1 Y_1^2, \quad B = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$$

and

$$X_3 = B^2 - 8A, \quad Y_3 = -8Y_1^4 + B(4A - X_3), \quad Z_3 = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2.$$

One can verify that these two operations can be performed using 11M+5S and 3M+5S respectively. It has been shown that some of the intermediate results can be reused in specific cases. More precisely, if a point $P = (X_1, Y_1, Z_1)$ is added to any other point, it is possible to store the data Z_1^2 and Z_1^3. During the same scalar multiplication, if the point P is added again to another point, reusing those stored values saves 1M+1S. This is what is usually called a readdition and its cost is 10M+4S instead of 11M+5S. With mixed and, of course, the general addition (one of the added points has its z-coordinate equal to 1), this is the only kind of point additions that can occur in all the traditional scalar multiplication methods.

Our new method allows more variety in caching strategies and point addition situations. From the doubling formulas, we can see that if we store Z_1^2 after the doubling of P and if we have to add P to another point, reusing Z_1^2 saves 1S. Adding a point that has already been doubled will be called $dADD$.

We now apply this to our scalar multiplication algorithm. We first compute the sequence $P \to 2P \to \cdots \to 2^{b_{max}}P$. For each doubled point (i.e., $P \to 2P \to \cdots \to 2^{b_{max}-1}P$), it is possible to store Z^2. Different situations can now arise:

- **addition after doubling (dADD)**: addition of a point that has already been doubled before
- **double addition after doubling (2dADD)**: addition of two points that have already been doubled before
- **addition after doubling + readdition (dreADD)**: addition of a point that has already been doubled before to a point that has been added before

Curve shape	dADD	2dADD	dreADD	2reADD	dmADD	mreADD
3DIK	11M+6S	11M+6S	10M+6S	9M+6S	7M+4S	6M+4S
Edwards	10M+1S	10M+1S	10M+1S	10M+1S	9M+1S	9M+1S
ExtJQuartic	7M+3S	7M+2S	7M+2S	7M+2S	6M+2S	6M+2S
Hessian	6M+6S	6M+6S	6M+6M	6M+6S	5M+6S	5M+6S
InvEdwards	9M+1S	9M+1S	9M+1S	9M+1S	8M+1S	8M+1S
JacIntersect	11M+1S	11M+1S	11M+1S	11M+1S	10M+1S	10M+1S
Jacobian	11M+4S	10M+4S	10M+3S	9M+3S	7M+3S	6M+3S
Jacobian-3	11M+4S	10M+4S	10M+3S	9M+3S	7M+3S	6M+3S

TABLE II
NEW ELLIPTIC CURVE OPERATIONS COST

- **double readition (2reADD)**: addition of two points that has been added before
- **addition after doubling + mixed addition dmADD**: addition of a point that has already been doubled before to a point in affine coordinate (i.e., $Z = 1$)
- **mixed readdition (mreADD)**: addition of a point in affine coordinate (i.e., $Z = 1$) to a point that has been added before.

Our caching strategies provide a classical time-memory trade-off. It adds one more coordinate to be stored for each stored point and saves on average one squaring per addition. For instance, considering a point in Jacobian coordinates, storing the Z^2 increases the memory requirement by 25% and saves between 1% to 2% computation time.

Remark 6.1: It is also possible to cache Z^2 after a tripling. Adding a point whether it has already been doubled or tripled has the same cost. Thus, we will still call this operation **dADD**.

In Table II we summarize the costs of different operations for each considered curve.

B. Window greedy algorithm performance

We have implemented the previous algorithms in C with `gmp` library to manipulate large numbers, compiled with gcc 4.6. We have measured the time and expansion sizes in various situations for 1000 random integers of sizes 192, 224 and 256 bits on a 3.30GHz Intel Core i5-2500 CPU. The results are summarized in Tables III, IV and V. For comparison purposes, we also have implemented the DBC conversion method from [23]. It is in fact a multi-base version of the traditional binary decomposition algorithm and thus performs really fast. However, the

need for 3 divisibility tests prevents the use of our window approach to speed up the process. In our experiment, we have considered a window size of 5, as this is the optimal choice in terms of performance for scalar multiplication using 192- to 256-bit scalars.

Results clearly show the benefit of using a windowed version of the greedy algorithm. Manipulating only register size operands allows us to compute DB expansions up to 15 times faster than with the basic greedy algorithm. Moreover, in the context of a fixed exponent, it becomes realistic to use the depth 2 or 3 version of the greedy algorithm. Depth 2 window greedy algorithm behaves similar to the basic depth 1 algorithm in terms of speed, but provides significantly sparser expansions. Going for depth 3 allows us to obtain even sparser expansions (up to 10%) but tends to become a lot more time consuming. Still in the context of fixed exponent, it would be realistic to go for a larger depth, however the gain tends to become drastically smaller compared to the amount of time spent. That is why we have limited our study to depth 3.

b_{max} / t_{max}	alg.	exp. size	time (μs)
192 / 38	w-greedy(1)	25.52	12.11
	w-greedy(2)	23.87	196.69
	w-greedy(3)	22.95	9399.01
	greedy	25.52	153.04
132 / 38	w-greedy(1)	27.65	10.98
	w-greedy(2)	26.16	177.80
	w-greedy(3)	25.64	7518.61
	greedy	27.65	151.96
	db chains	28.96	7.2

TABLE III

COMPARISON OF SPEED AND EXPANSION SIZE OF THE VARIOUS GREEDY ALGORITHMS FOR 192-BIT INTEGERS

b_{max} / t_{max}	alg.	exp. size	time (μs)
224 / 38	w-greedy(1)	29.50	14.01
	w-greedy(2)	27.61	233.96
	w-greedy(3)	26.49	11281.05
	greedy	29.50	183.66
164 / 38	w-greedy(1)	31.66	13.45
	w-greedy(2)	29.94	217.18
	w-greedy(3)	29.23	9424.95
	greedy	31.66	179.66
	db chains	33.78	8.6

TABLE IV

COMPARISON OF SPEED AND EXPANSION SIZE OF THE VARIOUS GREEDY ALGORITHMS FOR 224-BIT INTEGERS

b_{max} / t_{max}	alg.	exp. size	time (μs)
256/ 38	w-greedy(1)	33.38	14.31
	w-greedy(2)	31.21	270.86
	w-greedy(3)	30.00	13055.43
	greedy	33.38	214.90
196 / 38	w-greedy(1)	35.42	14.01
	w-greedy(2)	33.51	254.62
	w-greedy(3)	32.72	11269.38
	greedy	35.42	216.96
	db chains	38.38	10.3

TABLE V

COMPARISON OF SPEED AND EXPANSION SIZE OF THE VARIOUS GREEDY ALGORITHMS FOR 256-BIT INTEGERS

C. Performance analysis

We have carried out experiments on 192-, 224- and 256-bit scalars over all the elliptic curves mentioned in section II-A and all values of b_{max} and t_{max} such that $2^{b_{max}}3^{t_{max}}$ is a 192, 224 or 256-bit integer. For each curve and each set of parameters, we have:

- generated 10000 pseudo random integers in $\{0, \ldots, 2^m - 1\}, m = 192, 224$ and 256,

- converted each integer into the DB number system using the corresponding parameters,
- counted all the field multiplications (M) and squarings (S) involved in the point scalar multiplication process.

To compare with DBC, we have included results from [19] for 256-bit integers and used our own implementation for other scalar sizes. Results for Extended Jacobi Quartics and Hessian curves slightly differ from [19] since faster point composition formulae have been proposed since the publication of the article. Table VI summarizes the results of our experiments. For the window greedy algorithms, we have added, between parenthesis, the optimal choice for t_{max}, i.e. the value that led to the best average result. To ease the task of comparisons with previous works, we made the classical assumption $S = 0.8M$. However, different ratios could give slightly different results. It clearly shows that CDB expansions are superior to DBC in almost every situation. In particular, the best results are obtained on Extended Jacobi Quartic curves, for which we perform about 4% faster than DBC. As expected, if the scalar is fixed and one can spend some time on precomputation, using the depth 2 or 3 version of the window greedy algorithm allows some additional savings making our approach about 8% faster than its chained counterpart.

Curve shape	w-greedy(1)	w-greedy(2)	w-greedy(3)	DB chain
3DIK 192	1817.6 (38)	1796.2 (37)	1786.2 (35)	1810.0
3DIK 224	2123.1 (38)	2098.1 (37)	2084.7 (36)	2101.1
3DIK 256	2428.0 (38)	2397.2 (37)	2382.0 (36)	2393.2[(1)]
Edwards 192	1535.3 (12)	1517.9 (16)	1507.3 (18)	1581.1
Edwards 224	1783.8 (18)	1762.0 (18)	1748.8 (20)	1830.5
Edwards 256	2030.0 (20)	2004.1 (22)	1990.6 (20)	2089.7[(1)]
ExtJQuartic 192	1466.5 (12)	1453.6 (12)	1447.0 (12)	1526.3
ExtJQuartic 224	1704.1 (12)	1688.4 (12)	1678.6 (16)	1769.3
ExtJQuartic 256	1941.0 (11)	1922.0 (14)	1910.2 (15)	2021.2[(1)]
Hessian 192	1794.8 (36)	1778.5 (36)	1768.3 (33)	1825.1
Hessian 224	2087.4 (36)	2067.4 (35)	2057.9 (31)	2121.8
Hessian 256	2379.7 (36)	2356.7 (36)	2345.5 (36)	2420.1[(1)]
InvEdwards 192	1505.4 (12)	1490.7 (16)	1481.7 (18)	1542.6
InvEdwards 224	1750.7 (16)	1731.1 (18)	1719.7 (20)	1788.1
InvEdwards 256	1993.0 (20)	1970.1 (22)	1957.5 (20)	2041.2[(1)]
JacIntersect 192	1550.8 (11)	1535.7 (12)	1525.8 (13)	1713.3
JacIntersect 224	1800.2 (11)	1780.6 (12)	1768.5 (12)	1986.8
JacIntersect 256	2048.1 (11)	2026.1 (15)	2010.2 (15)	2266.1[(1)]
Jacobian 192	1829.3 (30)	1806.9 (29)	1793.7 (30)	1859.5
Jacobian 224	2124.4 (30)	2097.9 (31)	2083.7 (30)	2157.9
Jacobian 256	2417.2 (36)	2387.3 (34)	2369.4 (30)	2466.2[(1)]
Jacobian-3 192	1759.9 (28)	1735.7 (27)	1721.9 (24)	1795.6
Jacobian-3 224	2043.4 (29)	2014.7 (25)	1998.6 (25)	2084.0
Jacobian-3 256	2323.3 (37)	2292.4 (30)	2274.8 (29)	2379.0[(1)]

(1) Bernstein et al. [19]

TABLE VI

FIELD MULTIPLICATION COUNT FOR 192, 224 AND 256 BIT SCALARS AND VARIOUS DOUBLE-BASE EXPANSIONS

D. Comments

For high performance of cryptographic operations, many researchers have proposed dedicated hardware to accelerate the field arithmetic, especially multiplication. For example, the work in [24] (p. 41) reports that a 192-bit modular multiplication can be performed in 14 cycles in ASIC with a clock frequency of 138 MHz, i.e., one multiplication can be performed in only $14 * 10^{-6}/138 = 0.10145$ μs. Using such a hardware unit for both field multiplication and squaring, a

192-bit ECC scalar multiplication on ExtJQuartic would require about 1672.19×0.10145= 169.64 μs. Expansion of the scalar k into the DB representation however cannot easily take advantage of the field multiplier and hence is expected to rely on a general purpose processor. Thus the time for the expansion of k can be close, or of the same order, to that of the scalar multiplication in the hybrid hardware-software implementation of cryptographic systems. When k is not known in advance and needs to be re-coded into DB representation on-the-fly, our proposed expansion method can provide considerable speed-up for the overall scalar multiplication operation. As an example, a 192-bit scalar expansion (for which some timing results are given in Table III) followed by a scalar multiplication on ExtJQuartic will take about 353.3 μs using the greedy algorithm, but only 181.75 μs using the proposed w-greedy(1) algorithm.

VII. Conclusions

In this paper we have proposed efficient algorithms to take advantage of double-base expansions in the context of elliptic curve scalar multiplication. We have proposed a generalized version of Yao's algorithm, along with a constrained double-base representation that is less restrictive than the double-base chain. The main advantage of this representation is that it takes advantage of the natural sparseness of the double-base number system without any additional and unnecessary computations. We have proved that, despite its constraints, our representation still behave asymptotically like the general double-base expansion. We have introduced a window version of the greedy algorithm enabling the computation of the double-base expansion up to 15 times faster. Finally, our experiments show that our method performs generally faster than its chained counterpart for over all types of curves considered here.

References

[1] V. Dimitrov and T. Cooklev, "Two algorithms for modular exponentiation using nonstandard arithmetics," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 78, no. 1, pp. 82–87, 1995.

[2] V. Berthé and L. Imbert, "On converting numbers to the double-base number system," in *Advanced Signal Processing Algorithms, Architecture and Implementations XIV*, ser. Proceedings of SPIE, vol. 5559. SPIE, 2004, pp. 70–78.

[3] C. Doche and L. Imbert, "Extended double-base number system with applications to elliptic curve cryptography," in *INDOCRYPT*, ser. LNCS, vol. 4329. Springer, 2006, pp. 335–348.

[4] V. Dimitrov, L. Imbert, and P. K. Mishra, "Efficient and secure elliptic curve point multiplication using double-base chains," in *ASIACRYPT*, ser. LNCS, vol. 3788, 2005, pp. 59–78.

[5] H. Cohen and G. Frey, Eds., *Handbook of Elliptic and Hyperelliptic Cryptography*. Chapman & Hall, 2006.

[6] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[7] C. Doche, T. Icart, and D. R. Kohel, "Efficient scalar multiplication by isogeny decompositions," in *Public Key Cryptography*, ser. LNCS, vol. 3958. Springer Berlin / Heidelberg, 2006, pp. 191–206.

[8] H. M. Edwards, "A normal norm for elliptic curves," *Bulletin of the American Mathematical Society*, vol. 44, pp. 393–422, 2007.

[9] D. J. Bernstein and T. Lange, "Faster addition and doubling on elliptic curves," in *Asiacrypt*, ser. LNCS, vol. 4833/2008, 2007, p. 2950.

[10] ——, "Inverted Edwards coordinates," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, ser. LNCS, vol. 4851. Springer Berlin / Heidelberg, 2007, pp. 20–27.

[11] D. V. Chudnovsky and G. V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," *Adv. Appl. Math.*, vol. 7, no. 4, pp. 385–434, 1986.

[12] H. Hisil, G. Carter, and E. Dawson, "New formulae for efficient elliptic curve arithmetic," in *INDOCRYPT*, ser. LNCS, vol. 4859, 2007, pp. 138–151.

[13] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson, "An intersection form for jacobi-quartic curves," Personal communication, 2008.

[14] S. Duquesne, "Improving the arithmetic of elliptic curves in the Jacobi model," *Inf. Process. Lett.*, vol. 104, no. 3, pp. 101–105, 2007.

[15] P.-Y. Liardet and N. P. Smart, "Preventing SPA/DPA in ECC systems using the Jacobi form," in *Cryptographic Hardware and Embedded Systems - CHES*. Springer-Verlag, 2001, pp. 391–401.

[16] P. K. Mishra and V. Dimitrov, "Efficient quintuple formulas for elliptic curves and efficient scalar multiplication using multibase number representation," in *Information Security*, ser. LNCS, vol. 4779. Springer Berlin / Heidelberg, 2007.

[17] P. Longa and A. Miri, "New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields," in *Public Key Cryptography*, ser. LNCS, vol. 4939. Springer Berlin / Heidelberg, 2008, pp. 229–247.

[18] D. J. Bernstein and T. Lange, "Explicit-formulas database," available at http://hyperelliptic.org/EFD.

[19] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters, "Optimizing double-base elliptic-curve single-scalar multiplication," in *Progress in Cryptology INDOCRYPT 2007*, ser. LNCS, vol. 4859/2007. Springer Berlin / Heidelberg, 2007, pp. 167–182.

[20] A. C. Yao, "On the evaluation of powers," *SIAM Journal on Computing*, vol. 5, no. 1, pp. 100–103, 1976.

[21] V. S. Dimitrov, G. A. Julien, and W. C. Miller, "An agorithm for modular exponentiation," *Information Procesing Letters*, vol. 66, pp. 155–159, May 1998.

[22] R. Tijdeman, "On the maximal distance between integers composed of small primes," *Compositio Mathematica*, vol. 28, no. 2, pp. 159–162, 1974.

[23] P. Longa and C. Gebotys, "Setting speed records with the (fractional) multibase non-adjacent form method for efficient elliptic curve scalar multiplication," in *Public Key Cryptography*, ser. LNCS, vol. 5443. Springer Berlin / Heidelberg, 2009, pp. 443–462.

[24] M. Knezevic, "Efficient hardware implementation of cryptographic primitives," Ph.D. dissertation, Katholieke Universiteit Leuven, 2011.