Optimizing Elliptic Curve Scalar Multiplication for small scalars

Pascal Giorgi[a] and Laurent Imbert[a,b] and Thomas Izard[a]

[a]LIRMM, CNRS, Université Montpellier 2
161 rue Ada, 34090 Montpellier, France;

[b]PIMS, CNRS, University of Calgary
2500 University Dr. NW, Calgary, T2N 1N4, Canada

ABSTRACT

On an elliptic curve, the multiplication of a point P by a scalar k is defined by a series of operations over the field of definition of the curve E, usually a finite field $\mathbb{F}_q$. The computational cost of $[k]P = P + P + \cdots + P$ (k times) is therefore expressed as the number of field operations (additions, multiplications, inversions). Scalar multiplication is usually computed using variants of the binary algorithm (double-and-add, NAF, wNAF, etc). If s is a small integer, optimized formula for $[s]P$ can be used within a s-ary algorithm or with double-base methods with bases 2 and s. Optimized formulas exists for very small scalars ($s \leq 5$). However, the exponential growth of the number of field operations makes it a very difficult task when $s > 5$. We present a generic method to automate transformations of formulas for elliptic curves over prime fields in various systems of coordinates. Our method uses a directed acyclic graph structure to find possible common subexpressions appearing in the formula and several arithmetic transformations. It produces efficient formulas to compute $[s]P$ for a large set of small scalars s. In particular, we present a faster formula for $[5]P$ in Jacobian coordinates. Moreover, our program can produce code for various mathematical software (Magma) and libraries (PACE).

Keywords: Elliptic curve cryptography, scalar multiplication, s-ary method, double-base chains

1. INTRODUCTION

Security of elliptic curve cryptography[1, 2] relies on the ECDLP, the elliptic curve version of the discrete logarithm problem. Over a group $(E(\mathbb{F}_q), +)$, where $E(\mathbb{F}_q)$ is the set of points of the curve E defined over the finite field $\mathbb{F}_q$, and $+$ denotes the group law, solving ECDLP consists in finding the scalar $k \in \mathbb{Z}$ given the points P and $Q = [k]P = P + P + \cdots + P$ (k times). The operation denoted $[k]P$ is called the point or scalar multiplication. The operation $P + Q$ with $P, Q \in E(\mathbb{F}_q)$ is defined by two series of operations over the field of definition $\mathbb{F}_q$; one for $P = Q$ (doubling) and one for $P \neq Q$ (addition). The scalar multiplication $[k]P$ for k of cryptographic size (more than 160 bits) is computed by classical algorithms such as double-and-add and NAF.[3] Both algorithms perform one doubling for each digit of the representation of k, plus one addition for each non-zero digit. NAF decreases the number of non-zero digits in the representation, and therefore reduces the number of additions. Others interesting ways to compute $[k]P$ are s-ary and double-base number system algorithms. While double-and-add uses binary representation of k, s-ary algorithm uses a radix-s representation of $k = \{k_l, \ldots, k_0\}_s$ with $k_i \in \{0, \ldots, s-1\}$ such that $k = \sum_i k_i s^i$. For each digit k_i, the s-ary operation $[s]P$ is computed and for each non-zero digit, $[k_i]P$ is added. This algorithm requires the precomputation of $[k_i]P$. Double-base number system (DBNS) is a number representation scheme where an integer is written as the sum of mixed powers of two co-prime numbers p and q. Let consider $p = 2$ and $q = s$. Then $k = \sum k_i 2^{a_i} s^{b_i}$ with $k_i \in \{-1, 1\}$. This representation has been used to speed-up exponentiation and scalar multiplication.[4] The efficiency of both the s-ary and the double-base algorithms relies upon the efficiency of the computation of $[s]P$, where s is a small scalar.

Further author information: Send correspondence to T. Izard: Thomas.Izard@lirmm.fr.
This work was funded by the Université Montpellier 2 and the Pacific Institute for the Mathematical Sciences (PIMS). This project was completed during Laurent Imbert and Thomas Izard's long term visits at the University of Calgary thanks to a PIMS-CNRS agreement and the "Conseil Scientifique" of the Université Montpellier 2 respectively.

Efficiency of $[s]P$ can be assessed in terms of field operations appearing in the computation: additions, multiplications and inversions. Inversion is the most expansive field operation, but projective coordinate systems introduce a third coordinate in the elliptic curve equation in order to avoid them. In the following we consider projective coordinate systems. The cost of group operations ($P+Q$, $[2]P$, $[3]P$, $[s]P$, *etc*) is therefore dominated by the number of field multiplications. In Table 1 we present the cost of each operation in term of multiplication equivalent. Example 1 presents the doubling formula in Jacobian coordinates. It is extracted from the explicit formulas database.[5]

Table 1. Cost of different operations over the field

Operation	Notation	Cost
Multiplication	M	$1M$
Square	S	$0.67M \leq S \leq 0.8M$
Multiplication by small constant c	cM	Depends on the size of c
Addition-Subtraction	A	Negligible

Example 1. *In Jacobian coordinate system, a doubling operation costs* $3M + 6S + 6cM + 4A$*, as the point* $[2]P = (X_3, Y_3, Z_3) = [2](X_1, Y_1, Z_1)$ *is given by:*

$$\begin{aligned} X_3 &= (3 \times X_1^2 + a \times Z_1^4)^2 - 8 \times X_1 \times Y_1^2 \\ Y_3 &= (3 \times X_1^2 + a \times Z_1^4) \times (4 \times X_1 \times Y_1^2 - X_3) - 8 \times Y_1^4 \\ Z_3 &= 2 \times Y_1 \times Z_1 \end{aligned}$$

Multiplications, squares, additions and subtractions are performed with operands of size at least 160 bits. The multiplications by small constants c are special cases; their cost depends on the size of c. In example 1, a represents a parameter of the curve, which can be chosen in $\mathbb{F}_q$ as a 160 bits number or can be small. In particular, choosing $a = -3$ allows a transformation reducing the cost of the formula.

Optimizing the computation $[s]P$ is achieved by decreasing the number of operations over the field. By hand, the optimizations are difficult, even impossible, for $s > 5$ because of the number of operands and operations, and the number of possible arithmetic transformations. Our solution consists in automating the optimizations. Thanks to directed acyclic graph we first eliminate common-subexpressions appearing in the computations. Then we apply some arithmetic transformations in order to increase the efficiency of the multiplication $[s]P$. We conjecture this problem of optimization is NP-complete, but simple heuristics work fine for small scalar s. In particular, we propose a new formula to compute $[5]P$ in Jacobian coordinates system.

The rest of the paper is organized as follows. In section 2 we define the problem and its complexity. Then, in sections 3 and 4 we present our heuristic and our scheme to automate the optimization. Section 5 shows our experimental results.

2. THE FORMULA OPTIMIZATION PROBLEM

Definition 2.1. *We call formula, noted F, the set of operations corresponding to the computation of $[s]P$. The cost of a formula, noted $C(F)$, is the sum of the costs of each operation (see table 1). We say that two formulas are equivalent (noted $F \sim G$) if they give the same results for identical inputs.*

Definition 2.2. *In a formula F, a term can be either a variable, a constant, or an operator.*

Example 2. *Let $\lambda = 3X^2 + aZ^4$ be a formula. It can be represented by a tree as shown in Figure 1. The formula λ has ten terms: two variables (X and Z), two constants (a and 3), and six operators ($+$, $\times$, $\times$, ˆ2, ˆ2, ˆ2).*

Given a formula F which computes $[s]P$, the formula optimization problem consists in finding an optimal formula $G \sim F$ computing $[s]P$, *i.e.* such that $\forall F_i \sim F$, $C(G) \leq C(F_i)$.

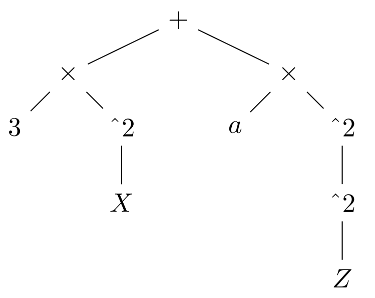

Figure 1. The tree representing the 10-term formula $\lambda = 3X^2 + aZ^4$

For any s, the double-and-add or NAF algorithms can be used to produce a base formula F. Given such a base formula, several transformations can be performed over it: first identify all common subexpressions so that the same calculation is only performed once, then apply transformations in order to further reduce the number of field operations, mainly multiplications. The formula optimization formula is clearly a difficult problem. We conjecture it to be NP-complete for reasons given below.

Due to classical algorithms, the size of formulas grows exponentially following two parameters: the size and the Hamming weight of the representation of s. Table 2 presents the number of terms (operators, variables and constants) appearing in formulas to compute $[s]P$ in Jacobian coordinate.

Table 2. Size of formulas in Jacobian coordinates system

Algorithm	$[2]P$	$P+Q$	$[3]P$	$[7]P$	$[13]P$	$[17]P$
Double-and-add	98	204	953	71898	611318	564585
NAF	98	204	8005	67811	5066371	564585

For $[3]P$, the formula generated by NAF is more than 8 times larger than that computed using the Double-and-add algorithm. In both NAF and binary representations, 3 contains two non-zero digits, but NAF representation has one more digit than binary representation. This explains the gap between NAF and double-and-add formulas for $[13]P$. Reversly, even if the NAF representation of 7 is larger than the binary, it contains one less non-zero digit. Note that the formulas to compute $[17]P$ are shorter in both cases than those to compute $[13]P$. This is due to the fact that 17 has only two non-zero digits in its representation. This table illustrates the first problem of the optimization: the sizes of the formulas to optimize are huge, even for small scalars.

3. OPTIMIZATION METHODS

Let consider the binary tree $T = (X, E)$ corresponding to the formula F, where internal nodes of T are the operators and the leaves are the variables and the constants. We call arithmetical cost of T, denoted $C(T)$, the sum of the costs associated to each operator, then $C(T) = C(F)$. The first step to optimize the formula is to eliminate its common subexpressions, *i.e.* expressions in the formulas leading to the same result. In order to find these common subexpressions, it is necessary to take into account associativity, commutativity and distributivity properties. Directed acyclic graph (DAG) is a graph in which each subtree appears only one time. By transforming the tree T in such a DAG T', each subexpression of F appears only once, and then $C(T') \leq C(T)$.

Computing the *minimal* directed acyclic graph consists in finding the expressions, in the tree, which have same result. Each internal node is a root of a subexpression which can be written as a polynomial in X, Y and Z, as illustrated in example 3. The terms of this polynomial can be sorted by the power of (X, Y, Z), and then two polynomials can be compared. Due to the size of the formula, this method requires exponential storage to keep coefficients and powers of the polynomial for each node, and is therefore not practical. In section 4, we propose a heuristic to address this problem.

Example 3. *Let us consider the subexpression*

$$E = \left(3 \times X_1^2 + a \times Z_1^4\right) \times \left(4 \times X_1 \times Y_1^2 - \left(\left(3 \times X_1^2 + a \times Z_1^4\right)^2 - 8 \times X_1 \times Y_1^2\right)\right)$$

which appears in the doubling formula (see example 1). The polynomial corresponding to this expression is

$$P(E) = -27X_1^6 + 36X_1^3Y_1^2 - 27aX_1^4Z_1^4 - 9a^2X_1^2Z_1^8 + 12aX_1Y_1^2Z^4 - a^3Z_1^12.$$

Figure 2 represents the DAG of the doubling formula in Jacobian coordinates (this graphic has been created with Tulip,[6] a GNU software from the LaBRI, Bordeaux, France). Remark that none of the subtrees of the DAG appear more than once. Note that the DAG is still a binary tree from roots to leaves.

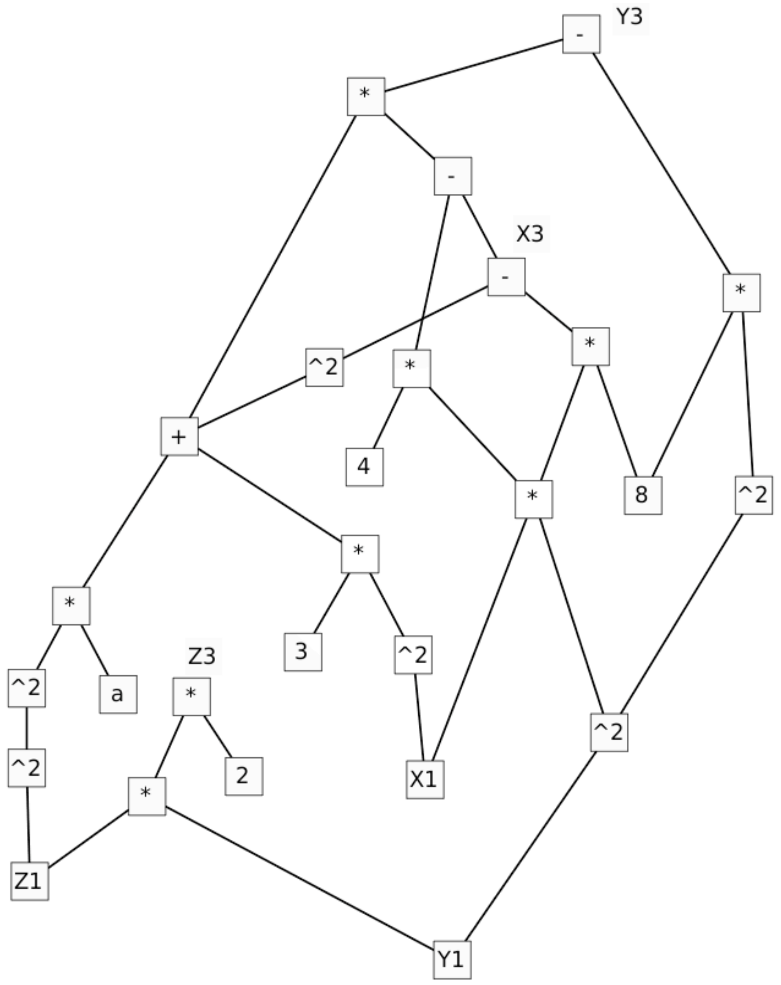

Figure 2. DAG of doubling formula in Jacobian Coordinates System

Main matter of the building of the minimal DAG is set by the associative and commutative operations (addition and multiplication). Question is, given a set of multiplications (or addition), finding the minimal number of products (or sums) needed to compute all the operations of the set. This problem is called ensemble computation and is known to be NP-complete (see [7, p. 66]). Example 4 illustrates this problem in an arbitrary case: how to compute a set of four multiplications chains with the smallest number of products.

Example 4. *Let consider four chains of multiplications: aabcd, aab, abc, and ad. The problem is to find the minimal number of products to compute all the multiplications. Clearly ab has to be compute as it appears in* *three chains. The chain aab has to be computed, and can be used for the computation of the first chain, but third chain multiplied by fourth is equal to the first. Then, the minimal number of products to compute the four chains is five: $A = ab$, $B = aA = aab$, $C = Ac = abc$, $D = ad$ and $E = CD = abcad$.*

Let consider that we can compute the minimal DAG (minimal in terms of nodes). Each internal node of the DAG is an operation which appears only once. We can now apply some transformations over this graph to decrease its arithmetical cost. In projective coordinates system multiplication and square are the most expansive operations. It is widely admitted that the cost of a square is between 0.67 and 0.8 times that of a multiplication. With field identities, we can easily transform a multiplication in sum of squares,[8] or subtraction of squares in multiplication.

$$\begin{aligned} ab &= \frac{1}{2}((a+b)^2 - a^2 - b^2) && (1)\\ ab &= \frac{1}{2}(a^2 + b^2 - (a-b)^2) && (2)\\ ab &= \frac{1}{4}((a+b)^2 - (a-b)^2) && (3)\\ a^2 - b^2 &= (a+b)(a-b) && (4) \end{aligned}$$

The transformation of multiplications in sum of squares by equations (1) and (2) will only be applied if two of the three squares are needed elsewhere in the formula, because replacing the multiplication by more than one square will be more expansive than the multiplication itself. Equation (3) needs only one square. Equation (4) can be applied to transform a multiplication into a subtraction of squares, but if only the two squares (a^2 and b^2) are needed in the computation, it is more interesting to perform a multiplication.

Example 5. *Let consider the doubling formula introduced in example 1. Since X^2, Y^2, Y^4 and Z^2 have to be computed in the formula, the two multiplications $X_1Y_1^2$ and Y_1Z_1 can be transformed by* (1) *(or* (2)*).*

Reducing the formula F to a minimal DAG also solves the problem of the minimal number of multiplication (and addition) to compute a set of products (or sums) which is known to be NP-complete. For that reason, we conjecture the formula optimization problem to be NP-complete. However, the heuristic presented below allows to construct a DAG and then apply some transformations to reduce its arithmetical cost.

4. HEURISTIC AND IMPLEMENTATION

In this section we present our general method to optimize formulas automatically. This method was implemented in C++. Due to the complexity of the problem, we first build the DAG using the heuristic presented in 4.1, then all multiplications are tested to be transformed into sum of square.

4.1 Elimination of common subexpressions

The problem of the common subexpressions elimination is set by the associative and commutative operations. As we saw in section 3, the storage for each node of coefficients and powers of its equivalent polynomial is exponential. Our heuristic consists in evaluating, for each node, its polynomial at sufficiently many point. At the beginning of the program, we randomly choose a list of values for X, Y, Z and a. During the construction of the DAG, for each new node n we evaluate the polynomial rooted by n, modulo a 31-bit prime number (to avoid multiple precision arithmetic). Each child of n has the list l_i of its own evaluations. We compute the operation n with as operands the list(s) of values of the child(ren) of n and we obtain a new list l_n. Then, we compare l_n with the list of each node already known in the DAG. If we find a node n' which has the same list l_n, the node n is replaced by n', else we add the new node n to the DAG. We choose to evaluate the polynomials at 1000 random points. The probability of errors is very small. Since our program also produces Magma scripts for verification, these possible errors can be easily found. Example 6 shows the construction of the DAG of an expression with small numbers.

Example 6. *Let us consider the expression*

$$E = \left(3X_1^2 + aZ_1^4\right) \times \left(4X_1Y_1^2 - \left(\left(3X_1^2 + aZ_1^4\right)^2 - 8X_1Y_1^2\right)\right)$$

used in Example 3. Table 3 gives the randomly chosen values for a, X_1 and Z_1. We choose as prime number $p = 101$.

Table 3. Values for a, X_1 and Z_1

a	5	15	34	23	12	10
X_1	34	56	23	72	39	87
Z_1	20	45	98	47	14	68

We parse the formula from left to right, starting by the deepest term. We begin by evaluating the left subexpression: 3 then X_1, then X_1^2, and so on. Table 4 gives the evaluation of each node of the subexpression $(3X_1^2+aZ^4)$. (The constants and variables are not evaluated since they are already known). None of these expressions is known in the DAG, so each is added to the DAG (figure 3(a)).

Table 4. Values of first nodes of the DAG

1	$X_1^2 \bmod p$	45	5	24	33	6	95
2	$3X_1^2 \bmod p$	34	15	72	99	18	83
3	$Z_1^2 \bmod p$	97	5	9	88	95	79
4	$Z_1^4 \bmod p$	16	25	81	68	36	80
5	$aZ_1^4 \bmod p$	80	72	27	49	28	93
6	$(3X_1^2 + aZ_1^4) \bmod p$	13	87	99	47	46	75

The formula is parsed and the expression aZ_1^4 is now evaluated. This node is already present in the DAG. Same evaluation is made for the expression $3X_1^2$. Both nodes are shared (see figure 3(b)). Then, we evaluate the expression $(aZ_1^4+3X_1^2)$. This list is known in the DAG. It corresponds to the node 6 which is shared. Figure 3(c) shows the DAG of the expression E.

Thanks to this heuristic, we are able to find common subexpressions appearing in the DAG, and to compute them only once. The only nodes that cannot be shared are the nodes corresponding to the NP-complete problem presented in previous section. But, by construction, this problem appears very rarely.

4.2 Reducing the arithmetical cost

Once the DAG has been created, we apply some transformations to reduce its arithmetical. We use field identities to transform multiplications into sum of squares if it is possible. With field identities we consider only multiplications involving two operands, but as the multiplication is associative and commutative, in case of chain of multiplication (*abc...*) we have to test all the possible combinations, by considering that a multiplication of the chain is an operand if it appears elsewhere in the computation. Example 7 shows the number of tests we have to perform in each case of a four operands multiplications chain.

Example 7. *Let consider the chain abcd. After DAG construction, four cases can appear:*

- *abcd (Figure 4(a)), the multiplication is totally free, we have to test six cases (ab, ac, ad, bc, bd, cd);*

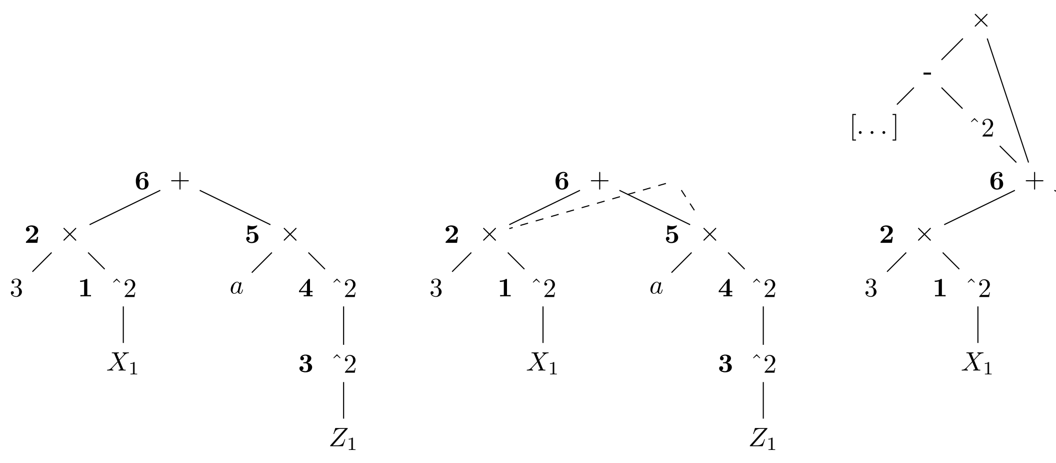

Figure 3. Construction of the DAG

- *$a(bcd)$ (Figure 4(b)). This case corresponds to a two-operand multiplication: the multiplication bcd appears elsewhere in the DAG. This multiplication bcd will be tested independently. Then, we just have to test the multiplication $a(bcd)$;*
- *$ab(cd)$ (Figure 4(c)): multiplication cd cannot be broken. We have to test three multiplications: $a(cd)$, $b(cd)$, ab. Note that $ab(cd)$ cannot be transformed since we work on the DAG: ab appears inevitably only once in the DAG: therefore it cannot appear in a square.*
- *$(ab)(cd)$ (Figure 4(d)): corresponds to a two-operand multiplication, we just test $(ab)(cd)$. The multiplications ab and cd will be tested independently.*

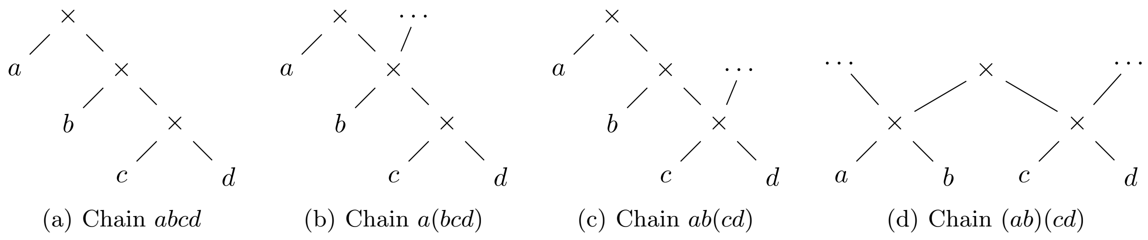
Figure 4. Reducing the cost of the multiplication chain $abcd$

We try to transform each two-operand multiplication into a sum of squares with field identities. Let consider $m = a \times b$. Three cases have to be taken into account:

- $(a+b)^2$ and $(a-b)^2$ are both needed in F. In this case, this multiplication is transformed by (3), and the new square is added to square set;
- $(a+b)^2$ or (exclusive) $(a-b)^2$ and a^2 or b^2 are needed in F. In this case, we chose (1) or (2): transformation by (3) is less expansive, in the sense that (3) only needs one subtraction, contrary to (1) and (2) which requires one subtraction and one addition. Point is that if only one square has to be created, $(a-b)^2$ (or $(a+b)^2$) would serve only for the computation of $a \times b$ whereas computing a new square a^2 (or b^2) can serve for other multiplications where a (or b) appears. In this case, all the multiplications have to be tested again.
- a^2 and b^2 are known in F: transform m by (1) or (2), including case where one of the addition/subtraction square is known.

Remark: Note that subtraction is not commutative. Tests has to be performed for both $(a-b)^2$ and $(b-a)^2$.

Remark: Coefficients $\frac{1}{2}$ or $\frac{1}{4}$ appear in each transformations. These rational coefficients can be easily removed as the projective coordinates form an equivalence class: $(X,Y,Z) \approx (\lambda^c X, \lambda^d Y, \lambda Z)$. In case of Jacobian coordinate, $c=2$ and $d=3$.

Remark: Because of the minimal cost of square ($0.67M$), we can test each multiplication independently: in most cases, the transformation follow identities 1 or 2, and we have to know two of the three squares to transform. If the cost of the square is smaller, we have to consider group of multiplications, as illustrated in example 8.

EXAMPLE 8. *Let consider the two multiplications ab and bc, and that we know a^2 and c^2. If $S \geq 0.67$, we can not transform since the global cost of the transformation will be $3 \times 0.67 = 2.01$ (we have to compute three new squares: b^2, $(a+b)^2$, $(b+c)^2$). If $S \leq 0.66$, we cannot transform independently the multiplications: each needs two new squares, but to transform the two multiplications, we need three new squares: $3 \times 0.66M = 1.98M$ which is cheaper than $2M$.*

The program takes as input the file containing the formula for basis operations (at least addition and doubling) and then compute the optimized formula for a given s. In the next section we present the experimental results in Jacobian coordinate system.

5. EXPERIMENTAL RESULTS

Thanks to our program, we were able to find a new formula to compute $[5]P$ in Jacobian coordinates. The quintupling formula presented in[9] (15M, 10S) was given as input to our program which found a new formula which only requires 7 multiplications and 16 squares. This formula has been validated with Magma. The algorithm is given below.

$$\begin{aligned}
A &= 3 \times X^2 + a \times Z^4 \\
B &= 6 \times ((X+Y^2)^2 - X^2 - Y^4) - A^2 \\
C &= (A+B)^2 - A^2 - B^2 - 16 \times Y^4 \\
D &= 4 \times Y^4 \times C \\
E &= 4 \times D - B \times B^2 \\
F &= (E - C^2) \\
G &= ((B+F)^2 - B^2 - F^2) \\
X_5 &= 4 \times (X \times E^2 - ((Y^2+C)^2 - Y^4 - C^2) \times G) \\
Y_5 &= 4 \times Y \times (B \times B^2 \times (E^2 + C^4 - 3 \times F^2) - 8 \times D \times C^4) \\
Z_5 &= (Z+E)^2 - Z^2 - E^2
\end{aligned} \tag{5}$$

Table 5 presents the experimental results for special computation in Jacobian coordinates system. Both $[2]P$ and $[3]P$ formulas are the best known formulas, extracted from the explicit formula database, and $[5]P$ is the formula (5). We can distinguish two main cases: the case of a single multiplication, $[s^i]P$ or $[s^it^j]P$, and the case of a multiplication followed by an addition. In the first case the program cannot optimize the formulas. This is not surprising since the same pattern is applied over larger and larger formula. There is no common subexpressions and the multiplications in the $[s]P$ formulas are already simplified. More interesting, is the case of a multiplication followed by an addition, we always save 2 multiplications in comparison to normal cost.

Our program generates code in several languages: generic pseudo-code, Magma code, or PACE code, and can easily generate code for others languages or mathematical program. With Magma code, we can test and certify our new formulas, using the same verification scripts as those presented in the explicit database.[5] The program generates the verification code using the Double-and-add algorithm to compute sure code. The Magma verification script for our quintupling formula is given below.

Table 5. Cost of formulas for $[k]$ in special case in Jacobian coordinates

$[k]P$	Old cost	New Cost
$2P$	1M+8S	-
$3P$	5M+10S	-
$5P$	15M+10S[9] 9M+15S[10]	7M+16S
$2^iP \pm 2^jP$ with $0 \leq j < i$	(i+11)M+(8*i+4)S	(i+9)M+(8*i+6)S
$3^iP \pm 3^jP$ with $0 \leq j < i$	(5*i+11)M+(10*i+4)S	(5*i+9)M+(10*i+6)S
$5^iP \pm 5^jP$ with $0 \leq j < i$	(15*i+11)M+(10*i+4)S	(7*i+9)M+(16*i+6)S
$2^i * 3^jP \pm 1$	-	(i+5*j+9)M+(8*i+10*j+6)S
$2^i * 5^j$	-	(i+7*j)M+(8*i+16*j)
$2^i * 5^j \pm 1$	-	(i+7*j+9)M+(8*i+16*j+6)
$3^i * 5^j$	-	(5*i+7*j)M+(10*i+16*j)
$3^i * 5^j \pm 1$	-	(5*i+7*j+9)M+(10*i+16*j+6)

```
K<a,b,X1,Y1> := FieldOfFrractions(PolynomialRing(Rationals(),4));
R<Z1> := PolynomialRing(K,1);

S := quo<R|Y1^2-X1^3-a*X1*Z1^4-b*Z1^6>;
x1 := X1/Z1^2; y1 := Y1/Z1^3;
S!(y1^2-x1^3-a*x1-b);
lambda := (3*x1^2+a)/(2*y1);
x2 := lambda^2-x1-x1; y2 := lambda*(x1-x2)-y1;
S!(y2^2-x2^3-a*x2-b);
lambda := (3*x2^2+a)/(2*y2);
x4 := lambda^2-x2-x2; y4 := lambda*(x2-x4)-y2;
S!(y4^2-x4^3-a*x4-b);
lambda := (y4-y1)/(x4-x1);
x5 := lambda^2-x1-x4; y5 := lambda*(x1-x5)-y1;
S!(y5^2-x5^3-a*x5-b);

AB := (16*((Y1)^2)^2);           AC := (a*((Z1)^2)^2);
AD := ((3*(X1)^2)+AC);           AE := (X1+(Y1)^2)^2;
AF := ((AE-(X1)^2)-((Y1)^2)^2);  AG := ((6*AF)-(AD)^2);
AH := ((AD+AG)^2-(AD)^2);        AI := ((AH-(AG)^2)-AB);
AJ := (((Y1)^2)^2*AI);           AK := ((4*(4*AJ))-(AG*(AG)^2));
AL := (AK-(AI)^2)^2;             AM := (AG+(AK-(AI)^2));
AN := ((AM)^2-(AG)^2);           AO := ((Y1)^2+AI)^2;
AP := (AO-((Y1)^2)^2);           AQ := ((AP-(AI)^2)*(AN-AL));
AR := ((X1*(AK)^2)-AQ);          AS := (((AI)^2)^2*(4*AJ));
AT := ((AK)^2+((AI)^2)^2);       AU := ((AG*(AG)^2)*(AT-(3*AL)));
AV := (Y1*(AU-(8*AS)));          AW := ((Z1+AK)^2-(Z1)^2);

X5 := (4*AR);
Y5 := (4*AV);
Z5 := (AW-(AK)^2);
S!(x5-X5/Z5^2); S!(y5-Y5/Z5^3);
```

This script first compute the symbolic set of points $[5]P$ for an arbitrary curve in Jacobian coordinate system. Then we compute $[5]P$ with the new formula, and the two sets are compared by the last instruction. If the result is $(0, 0)$, the sets are equal and the new formula is correct. Else, the formula computed by the program is wrong.

6. CONCLUSION

The problem of formula optimization is to reduce the number of field operations to compute point multiplication and is difficult, we conjecture NP-complete. By-hand, the optimization is very difficult, even impossible. We present a method to automate these optimizations, using simple and efficient heuristics to reduce the number of field multiplications: elimination of common subexpressions and transformations of multiplications into squares. Thanks to this method, we were able to optimize formulas for $[k]P$ for a large set of small scalar k in many coordinates systems. With our program, we produce code for mathematical softwares (magma) and for library (PACE). Moreover, the scripts produced for Magma allow the validations of the computed formulas.

REFERENCES

[1] Miller, V., "Use of elliptic curves in cryptography," in [*Advances in cryptology - CRYPTO 85*], *Lecture Notes in Computer Sciences*, 417–426, Springer (1985).

[2] Koblitz, N., "Elliptic curve cryptosystems," *Mathematics of Computation* **48**(177), 203–209 (1987).

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

[4] Dimitrov, V., Imbert, L., and Mishra, P. K., "The double-base number system and its application to elliptic curve cryptography," *Mathematics of Computation* **77**(262), 1075–1104 (2008).

[5] Bernstein, D. J. and Lange, T., "Explicit-formulas database," (2008). `http://www.hyperelliptic.org/EFD/index.html`.

[6] Auber, D. and Mary, P., "Tulip software." `http://tulip.labri.fr/`.

[7] Garey, M. R. and Johnson, D., [*Computer and Intractability : a guide to the theory of NP-Completeness*], W. H. Freeman (1979).

[8] Longa, P. and Miri, A., "Fast and flexible elliptic curve point arithmetic over prime fields," *IEEE Transactions on computers* **57**(3), 289–302 (2007).

[9] Dimitrov, V. and Mishra, P. K., "Efficient quintuple formulas for elliptic curves and efficient scalar multiplication using multibase number," in [*Information Security 10th International Conference*], *LNCS* **4779**, 390–406, Springer (2007).

[10] Longa, P. and Gebotys, C., "Setting speed records with the (fractional) multibase non-adjacent form method for efficient elliptic curve scalar multiplication." Cryptology ePrint Archive, Report 2008/118 (2008).