Efficient Regular Scalar Multiplication on the Jacobian of Hyperelliptic Curve over Prime Field Based on Divisor Splitting

Christophe Negre, Thomas Plantard

Efficient Regular Scalar Multiplication on the Jacobian of Hyperelliptic Curve over Prime Field Based on Divisor Splitting

Christophe Negre[2,3] and Thomas Plantard[1]

(1) CCISR, SCIT, University of Wollongong, Australia

(2) Team DALI, Université de Perpignan Via Domitia, France

(3) LIRMM, UMR 5506, Université de Montpellier and CNRS, France

Abstract

We consider in this paper scalar multiplication algorithms over a hyperelliptic curve which are immune against simple power analysis and timing attack. To reach this goal we adapt the regular modular exponentiation based on multiplicative splitting presented in JCEN 2017 to scalar multiplication over a hyperelliptic curve. For hyperelliptic curves of genus $g = 2$ and 3, we provide an algorithm to split the base divisor as a sum of two divisors with smaller degree. Then we obtain an algorithm with a regular sequence of doubling always followed by an addition with a low degree divisor. We also provide efficient formulas to add such low degree divisors with a divisor of degree g. A complexity analysis and implementation results show that the proposed approach is better than the classical Double-and-add-always approach for scalar multiplication.

Keywords. Hyperelliptic curve, regular scalar multiplication, divisor splitting, simple power analysis.

I. Introduction

Elliptic curve cryptography (ECC) was independently introduced by Koblitz [14] and Miller [22] in 1986-87. It is based on the intractability of the discrete logarithm problem in the group of points of an elliptic curve. The main advantage compared to the similar scheme over a group $(\mathbb{F}_q^*, \times)$ is that no subexponential algorithm is known which computes the discrete logarithm. The key size and the computation time are then reduced. Later Koblitz suggested in [15] its generalization (HECC) to the Jacobian of a hyperelliptic curve of genus g. Again for low genus there is no subexponential algorithm for the resolution of discrete logarithm problem. The advantage of using a hyperelliptic curve $H(\mathbb{F}_q)$ instead of an elliptic curve is that the cardinality of the Jacobian $Jac(H(\mathbb{F}_q))$ is close to q^g. This implies that for a given level of security the base field $\mathbb{F}_q$ is smaller and the operations like multiplication and addition are faster. But in counter parts the group law is more complicated than for an elliptic curve. Most of the research works done on HECC during the past 20 years was meant to improve the efficiency of the group operations.

Arithmetic of divisors on an hyperelliptic curve. The first approach to implement the arithmetic on a hyperelliptic curve is due to Cantor [3]. An element D of $Jac(H(\mathbb{F}_q))$ is a reduced divisor, i.e., it is a formal sum $D = \sum_{i=1}^{r}(P_i) - r(\infty)$ with $r \leq g$ where $P_i \in H(\mathbb{F}_q)$ and ∞ is the point at infinity. The approach of Cantor uses the Mumford representation of divisors on $J(H(\mathbb{F}_q))$: Mumford represents a divisor D with two polynomials $(U(X), V(X))$ such that the abscissas of the points P_i of the divisor are the roots x_i of $U(X)$ and the ordinates y_i of the points P_i are given by $V(x_i) = y_i$. The Cantor algorithm for divisor addition or doubling consists in a composition and a reduction steps, each consisting in a few small degree polynomial operations (multiplication, gcd and division). On genus 2 hyperelliptic curves the approach of Cantor was improved by Harley in [11] by expliciting the operations in $\mathbb{F}_q$ in order to minimize the overall number of field operations (multiplications and inversions) in the formulas. Later Lange pursued this idea in [18], [19] introducing a projective coordinate system in order to avoid field inversions and weighted projective coordinates to further reduce the complexity of the formulas. The last known improvement is due to Costello and Hisil in [12] which combines the co-Z approach (introduced by Meloni [21] over elliptic curves) and a new weighted projective coordinates system in order to further reduce the cost of divisor addition and doubling. We will consider also genus 3 hyperelliptic curves. For this type of curve the best results for affine coordinates are presented in [25]. In projective coordinates Fan *et al.* in [9] adapt the strategy of Lange [18], [19] to genus 3 curves: they provide explicit formulas non-weighted projective coordinate system over $Jac(H\mathbb{F}_p)$ and $Jac(H(\mathbb{F}_{2^m}))$.

For genus 2 hyperelliptic curves, another strategy takes advantage of Kummer surface to perform efficiently addition and doubling on the Jacobian. This approach was first used by Duquesne [8] and later a faster approach was provided by Gaudry in [10]. Gaudry proposes a formula for pseudo-addition (addition $D + D'$ where D, D' and the difference $D - D'$ are known) which is really efficient compared to arbitrary formulas in Mumford representation. One drawback of this approach is that we cannot perform usual divisor addition. This reduces the number of protocols which can be implemented with this approach: the Diffie-Hellman key exchange and a specific signature algorithm (cf. [10, Sect. 5.3]). Due to this limitation we will not consider this case in the sequel.

Side channel analysis. Implementation of cryptographic protocols can be threaten by side channel analysis. To get secret data, this type of attack monitors and analyses either computation time, power consumption, electromagnetic emanation, ... leaked out by the device performing the cryptographic computation. For example, the basic approach for scalar multiplication $K \cdot D$ of a divisor D is the Double-and-add algorithm consisting in a sequence of divisor doublings followed by an addition when the bit k_i of K is equal to one. Simple power analysis [17] monitors the power consumption and decompose the trace into a sequence of doubling and addition power traces. The eavesdropper can then deduce the sequence of the bits k_i since a bit is equal to 1 when a doubling is followed by an addition otherwise it is equal to 0. The timing attack [16] exploits the same weakness of the computation of $K \cdot D$ with the Double-and-add algorithm. Specifically, with a number of computation time samples, the timing attack extracts the key bits through a statistical analysis of these samples. Both SPA and timing attack can be defeated if we render the sequence of additions and doublings not correlated to the key bits. For example we can use the Double-and-add-always approach proposed by Coron in [6] which results a regular sequence of doublings always followed by an addition. More advanced attacks like differential power analysis [17] or horizontal attack [4], [27] require to randomize the representation of the group elements and the scalar. This can be done in addition to the regularity of the scalar multiplication (as the Double-and-add-always approach). In the sequel we will only focus on the prevention of the SPA and timing attack.

Contributions. In this paper we present an extension to HECC of the regular algorithm for modular exponentiation of [24]. The idea of [24] is to split the base element into two parts and modify the square-and-multiply algorithm in order to have all squaring followed by a multiplication with half-size element: this reduces the cost of the square-and-multiply-always approach. In the Jacobian of a hyperelliptic curve of genus 2 we rewrite the base divisor D of degree 2 as a difference of two degree one divisors $D = D_1 - D_0$. We provide an algorithm which performs this splitting. We can then process the scalar multiplication as a sequence of doublings always followed by an addition with a degree 1 divisor. We provide efficient formulas for these kind of addition in affine and weighted projective coordinates of Costello and Hisil [12]. Over genus 3 curves, we split the divisor D as a difference of two degree 2 divisors $D = D_1 - D_0$. We provide efficient formulas for addition of a degree 3 divisor with a degree 2 divisor based on the approach of Costello-Lauter [7]. For the case $g = 2$ and $g = 3$ we provide complexity comparison along with implementation results which show the benefit of the proposed approach compared to classical Double-and-add-always method.

Organization of the paper. In Section II, we first review the basics on Jacobian of hyperelliptic curve over a prime field and we recall algorithms used for regular scalar multiplication over hyperelliptic curves. In Section III we present an algorithm for divisor splitting and Add21 formula (addition of degree 2 with degree 1 divisor). We provide complexity comparison with best approach of the literature and implementation results. In Section IV we focus on genus 3 curves, following the same steps as for genus 2, we provide a divisor splitting algorithm, necessary addition formulas on the Jacobian, complexity used in the proposed regular scalar multiplication. and implementation comparisons. Finally, in Section V, we provide some concluding remarks.

II. Review of hyperelliptic curve and regular scalar multiplication approaches

In this section we first review the basics of the Jacobian of an hyperelliptic curve. We will then review regular algorithm for scalar multiplication.

A. Background on the Jacobian of hyperelliptic curve

We consider in this paper imaginary hyperelliptic curves of genus g over a prime field $\mathbb{F}_p$. They are given by an equation H of the form:

$$Y^2 = f(X) \text{ with } \deg f(X) = 2g + 1,$$

and has no singular point on the affine plane. Note that for a point $P = (x, y)$ on the curve, we denote $\bar{P} = (x, -y)$ which is also on the curve and we denote ∞ the point at infinity. The set of points on the curve plus the point at infinity is denoted $H(\mathbb{F}_p)$. Elements of the Jacobian $Jac(H(\mathbb{F}_p))$ are reduced divisors $D = \sum_{i=}^{r}(P_i) - r(\infty)$ with degree $r \leq g$ and $P_i \neq \bar{P}_j$ for $i \neq j$. A reduced divisor D can be set in the Mumford representation which defines a pair $u(X), v(X) \in \mathbb{F}_p[X]$ such that

$$\begin{array}{rcl} u(X) & = & \prod_{i=1}^{r}(X - x_i), \\ V(X)^2 - f(X) & \equiv & 0 \mod u(X), \end{array}$$

When all the points P_i are distinct, the condition on $V(X)$ means that $v(x_i) = y_i$ for $i = 1, \ldots r$. A group law is defined on $Jac(H(\mathbb{F}_p))$ as follows: for two reduced divisors $D = \sum_{i=}^{r}(P_i) - r(\infty)$ and $D' = \sum_{i=1}^{r'}(P_i') - r'(\infty)$ we first compose them $D'' = \sum_{i=}^{r} P_i + \sum_{i=1}^{r'} P_i' - (r + r')(\infty)$ and we remove the points P_i of D such that $\bar{P}_i$ appears also in D' (this is called the semi-reduction of D''). Note that in the specific case where $\gcd(u(X), u'(X)) = 1$ no semi-reduction is required, we just have to compute $u''(X) = u(X)u'(X)$ and $v''(X)$ such that

$$\begin{cases} v''(X) \mod u(X) = v(X), \\ v''(X) \mod u'(X) = v'(X). \end{cases}$$

In the general case we can have $\gcd(u(X), u'(X)) \neq 1$, in this case Cantor proposes in [3] to use Algorithm 1 for the composition and semi reduction of divisors.

Algorithm 1 Cantor algorithm for divisor composition [3]

```
**Require:** $D = (u(X), v(X)$ and $D' = (u'(X), v'(X))$ two reduced divisors
**Ensure:** $D'' = D + D'$ semi-reduced
  $d' \leftarrow \gcd(u, u') = eu + e'u'$
  $d \leftarrow \gcd(d', v + v') = cd' + c'(v + v')$
  $s \leftarrow ce, s' \leftarrow c'e', s_3 \leftarrow c'$
  $u'' \leftarrow uu'/d^2$
  $v'' \leftarrow \frac{suv' + s_2u'v + s_3(vv' + f)}{d} \mod u$
```

After the composition we can possibly have a divisor D'' with degree larger than g, i.e., $D'' = \sum_{i=0}^{r} P_i'' - r(\infty)$ with $r > g$. To reduce such divisor we subtract well chosen principal divisor (divisor of a rational function on the curve) which reduces the number of points in the divisors, i.e., the degree of $u''(X)$. When it is done a sufficient number a of time we get a reduced divisor, i.e., a divisor of degree $\leq g$. Cantor proposes in [3] Algorithm 2 which performs the reduction of a divisor in Mumford representation. For more details on hyperelliptic curves and Cantor algorithm the author may refer to [29], [5].

Cantor's approach is generic: it works over any field and for curves with any genus. Over genus 2 curves Harvey [11] optimizes this approach by expliciting the formula deduced from Cantor algorithm. A number of works followed this strategy: Lange [18], [19], Costello-Lauter [7] and Costello-Hisil [12] which further improved the complexity addition and doubling formula over genus 2 hyperelliptic curves. Over genus 3 curves, Fan *et al.* in [9] provides inversion free formulas and review the formulas of Nyukai *et al.* [25] for affine coordinates.

Algorithm 2 Cantor algorithm for reduction [3]

```
Require: $D'' = (u''(X), v''(X))$ semi-reduced divisor in $Jac(H(\mathbb{F}_p))$
Ensure: $D''' = (u''', v''')$ reduced satisfying $D''' \sim D''$
  $u''' \leftarrow u'', v''' \leftarrow v''$
  while $\deg u''' > g$ do
    $u''' \leftarrow \frac{f - v'''^2}{u'''}$
    $v''' \leftarrow -v''' \mod u'''$
  return $(u''', v''')$
```

B. Scalar multiplication algorithms

Given an integer K and an element $D \in Jac(H(\mathbb{F}_p))$, we would like to efficiently compute the scalar multiplication $K \cdot D$. A basic approach to compute this scalar multiplication $K \cdot D$ is the Double-and-add algorithm. This approach computes $K \cdot D$ with a sequence of doublings and additions in $Jac(H(Fd_p))$. This algorithm scans the binary representation of K and for each bit k_i it performs a doubling and an addition only when the bit k_i is equal to one. Unfortunately the Double-and-add approach is not secure against some side channel analyses. For example the simple power analysis can find the secret scalar K with a single power trace. If a doubling and an addition have different power traces, the power trace of a scalar multiplication can be decomposed as a sequence of doubling and addition power traces. The attacker can then deduce all the bits of the secret K.

In the literature there are several Double-and-add variants which are secure against SPA attack. The first one is the Double-and-add-always approach [6], where a dummy addition is performed when the bit is equal to 0. This approach is depicted in Algorithm 3. A second approach [13] uses a representation of K in base 2 with digits in $\{1, -1\}$. This representation is obtained by replacing the sequence of zeros in a binary representation $1000\ldots001$ by the equivalent expression $11\bar{1}\bar{1}\ldots\bar{1}\bar{1}\bar{1}$ where $\bar{1} = -1$. The scalar multiplication is computed with a sequence of doublings followed by either and addition or a subtraction. We will see later that on $Jac(H(\mathbb{F}_p))$ this approach is advantageous compared to the classical Square-and-multiply-always approach when some specific formula (co-ZW) are used for divisor operation.

Algorithm 3 Double-and-add-always

```
Require: $D \in Jac(H(\mathbb{F}_p))$ and $K = (k_{\ell-1}, \ldots, k_0)_2$
1: $R_0 \leftarrow \mathcal{O}, R_1 \leftarrow \mathcal{O}$
2: for $i$ from $\ell - 1$ downto $0$ do
3:     $R_1 \leftarrow 2 \cdot R_1$
4:     $R_{k_i} \leftarrow R_{k_i} + D$
5: return $R_1$
```

Remark 1. The Montgomery-ladder [23] is also a popular approach to counteract SPA attack since it is done

Algorithm 4 Signed-double-and-add-always

```
Require: $D \in Jac(H(\mathbb{F}_p))$ and $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{-1, 1\}$
1: $R \leftarrow \mathcal{O}, D_1 \leftarrow D, D_{-1} \leftarrow -D$
2: for $i$ from $\ell - 1$ downto 0 do
3:     $R \leftarrow 2 \cdot R$
4:     $R \leftarrow R + D_{k_i}$
5: return $R$
```

through a regular sequence of doubling always followed by an addition. As we will see later, we will use projective coordinates on hyperelliptic curves. These coordinates renders the Montgomery-ladder less efficient since it involves full projective additions while the Double-and-add-always (resp. Signed-double-and-add-always) algorithm is computed with mixed additions (resp. Co-Z additions) which are more efficient.

Scalar multiplication with divisor splitting. This approach is adapted from [24] it performs the exponentiation using a multiplicative half-size splitting of the base element. Over the Jacobian of an hyperelliptic curve we get regular scalar multiplication using an additive splitting of the base divisor. This method is depicted in Algorithm 5. This approach becomes more efficient than the above mentioned method of the literature (Double-and-add-always and Montgomery-ladder methods) if the additions with D_0 and D_1 are less expensive than a regular addition.

Algorithm 5 Regular scalar multiplication with additive splitting of the base divisor

```
Require: $D \in Jac(H(\mathbb{F}_p))$ and $K = (k_{\ell-1}, \ldots, k_0)_2$
Ensure: $r = k \cdot P$
1: Split. $D = D_1 - D_0$ for $D_0, D_1 \in Jac(H(\mathbb{F}_p))$.
2: $R \leftarrow -D_0$
3: for $i$ from $\ell - 1$ downto 0 do
4:     $R \leftarrow 2 \cdot R + D_{k_i}$
5: $R \leftarrow R + D_0$
6: return $r$
```

The following lemma establishes the validity of Algorithm 5, i.e., that it correctly computes $R = K \cdot D$.

Lemma 1. *Let $K = (k_{\ell-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$ be an ℓ-bit integer and let D be an element $Jac(H(\mathbb{F}_p))$. If we set $K_i = (k_{\ell-1}, \ldots, k_i)_2$, then the value of R after the i-th loop iteration satisfies:*

$$R = K_i \cdot D - D_0.$$

Proof. We prove the assertion of the lemma by a decreasing induction on i: we assume it is true for i and we prove it for $i - 1$. By induction hypothesis, R_i the value of R after the execution of loop i in Algorithm 5 satisfies

$R_i = K_i \cdot D - D_0$.

- If $k_{i-1} = 1$ the execution of loop $i-1$ gives: $R_{i-1} = 2 \cdot R_i + D_1 = (2K_i) \cdot D + (-2D_0) + D_1 = K_{i-1} \cdot D - D_0$.
- If $k_{i-1} = 0$, the execution of loop $i-1$ gives: $R_{i-1} = 2 \cdot R_i + D_0 = (2K_i) \cdot D + (-2D_0) + D_0 = K_{i-1} \cdot D - D_0$.

□

The goal of this paper is to render Algorithm 5 efficient on a hyperelliptic curve for the two practical cases genus 2 and 3 over a prime field $\mathbb{F}_p$.

III. Regular scalar multiplication on a genus 2 hyperelliptic curve

In the remaining of this section we focus on genus 2 hyperelliptic curves. Our goal is to have an efficient version of the regular algorithm with split base element (Algorithm 5). In the context of hyperelliptic curves of genus 2 this means that we have to split a divisor of degree 2 as a sum of two divisors of degree 1.

In the sequel we first explain how to perform the splitting of a degre 2 divisor for a genus 2 hyperelliptic curve $H(\mathbb{F}_p)$. We also provide formulas to add degree 1 divisor with a degree 2 divisor in affine and weighted projective coordinates to get an efficient version of Algorithm 5.

A. Splitting of divisors in $H(\mathbb{F}_p)$

Given a degree 2 divisor $D = (u(X), v(X))$ on $Jac(H(\mathbb{F}_p))$ of order N, we want to find two degree 1 divisors D_0 and D_1 such that

$$D = D_1 - D_0.$$

We can get this decomposition if we can factorize $u(X) = (X - x_0)(X - x_1)$ since then if we set

$$\begin{aligned} v_1 &= v(X) \mod (X - x_1), \\ v_0 &= -v(X) \mod (X - x_0), \end{aligned}$$

we obtain $D = D_1 - D_0$ with $D_1 = (X - x_1, v_1)$ and $D_0 = (X - x_0, v_0)$. Consequently, to split D the main problem is to factorize a degree 2 polynomial $u(X) = X^2 + u_1 X + u_0$ in $\mathbb{F}_p[X]$. To factorize such polynomial we compute the discriminant $\Delta = u_1^2 - 4u_0$ of $u(X)$. Then if Δ is a square in $\mathbb{F}_p$, we compute its square root $\delta = \sqrt{\Delta}$ and deduce the two roots of $u(X)$

$$x_0 = \frac{-u_1 + \delta}{2} \text{ and } x_1 = \frac{-u_1 - \delta}{2}.$$

If Δ is not a square then we cannot factorize $u(X)$ and we cannot split the divisor in $Jac(H(\mathbb{F}_p))$. In this case we compute $D' = 2 \cdot D$ and at the same time we divide the scalar by two $K' = K/2 \mod N$ (this is possible if $\gcd(2, N) = 1$). We get a new couple D', K' such that

$$K \cdot D = K' \cdot D'.$$

Then we can apply the same process to the new divisor D'. This approach is depicted in Algorithm 6.

Algorithm 6 Divisor splitting for a genus 2 hyperelliptic curve

```
Require: A degree 2 divisor $D = (u(X), v(X))$ of odd order $N$ and a scalar $K \in [0, N[$.
Ensure: two degree one divisors $D'_0, D'_1$ and $K'$ such that $D' = 2^t \cdot D$ and $D' = D'_1 - D'_0$ and $K' = K \times 2^{-t} \mod N$ for some $t \geq 0$.
 1: $D' \leftarrow D$
 2: $K' \leftarrow K$
 3: $\Delta' = u_1'^2 - 4u'_0$
 4: $s \leftarrow$ JacobiSymbol($\Delta'$)
 5: while $s = -1$ do
 6:     $D' \leftarrow DoublingAff(D')$
 7:     $K' \leftarrow K' \times 2^{-1} \mod N$
 8:     $\Delta' = u_1'^2 - 4u'_0$
 9:     $s \leftarrow$ JacobiSymbol($\Delta'$)
10: $\delta' \leftarrow$ SquareRoot($\Delta'$)
11: $x'_0 \leftarrow \frac{-u'_1+\delta'}{2}, x'_1 \leftarrow \frac{-u'_1-\delta'}{2}$
12: $D'_0 \leftarrow (X - x'_0, v'_1 \times x'_0 - v'_0)$
13: $D'_1 \leftarrow (X - x'_1, -v'_1 \times x'_1 + v'_0)$
```

The cost of Algorithm 6 depends on the number of time we stay in the loop while. Indeed, in a loop iteration we have to compute a Jacobi symbol and perform a divisor doubling which are costly operations. The average number of loop iteration is given in the following lemma.

Lemma 2. *The average number of times we execute the while loop iteration in Algorithm 6 is equal to* 1.

Proof. The probability that $u(X)$ factorizes is equal to the probability that Δ is a square in $\mathbb{F}_p$. And since there are $(p-1)/2+1$ squares and $(p-1)/2$ non-squares we have

$$\alpha = \mathbb{P}[u(X) \text{ factorizes }] = \frac{(p-1)/2+1}{p} = \frac{1}{2} + \frac{1}{2p}.$$

and consequently the probability that $u(X)$ does not factorize is $1 - \alpha = \frac{1}{2} - \frac{1}{2p}$. Now the probability that we have to execute i times exactly the main loop in Algorithm 6 until we get a splitting is equal to $(1-\alpha)^i\alpha$. Then, if $\mathfrak{N}$ is the random variable corresponding number of time we compute stay in the loop while, we have the following expected value

$$\mathbb{E}[\mathfrak{N}] = \sum_{i=1}^{\infty} i \times \mathbb{P}[\mathfrak{N} = i] = \sum_{i=1}^{\infty} i(1-\alpha)^i\alpha.$$

For p sufficiently large we assume that $\alpha \cong 1/2$ and $1 - \alpha \cong 1/2$. Then replacing x by $1/2$ in the following identity of formal series:

$$\sum_{i=1}^{\infty} ix^{i+1} = \frac{x^2}{(1-x)^2}$$

leads to $\mathbb{E}[\mathfrak{N}] = 1$. $\square$

In Algorithm 6 there are two things we need to explicit to get an fully functional algorithm: the first one is the computation of the Jacobi symbol of Δ' which is equal to 1 if Δ' is a square and -1 otherwise, the second one is the computation of the square root of Δ'. We discuss these issues in the following subsections.

B. Computation of the Jacobi Symbol

There are two strategies to compute the Jacobi symbol $\binom{a}{p}$ of an element $a \in \mathbb{F}_p$: the first one is done with an exponentiation in $\mathbb{F}_p$ and the second one with an algorithm similar to the Euclidean algorithm.

- *Exponentiation approach.* This approach computes the Jacobi symbol using the following expression of the Jacobi symbol:
$$\binom{a}{p} = a^{(p-1)/2} \mod p.$$
It can be computed with, in average, $\log_2(p)$ squarings and $\frac{\log_2(p)}{2}$ multiplications in $\mathbb{F}_p$ (i.e., more precisely $\log_2((p-1)/2)$ squarings and $HW((p-1)/2)$ multiplications, where HW means Hamming weight) using the square-and-multiply algorithm.
- *Euclidean-like approach.* This method is due do Eisenstein and can be found in [28]. First, we split $\Delta' = 2^t \Delta''$ where Δ'' is odd then we have
$$\binom{\Delta'}{p} = \binom{2}{p}^t \binom{\Delta''}{p}.$$
For the term $\binom{2}{p}^t$, the value $\binom{2}{p}$ can be precomputed and this leads to $\binom{2}{p}^t = \binom{2}{p}^{t \mod 2}$. We need now to compute $\binom{a}{b}$ where a and b are two odd integers (here $a = \Delta''$ and $b = p$). In this case the reciprocity law provides that
$$\binom{a}{b} = (-1)^{\frac{a-1}{2}\frac{b-1}{2}} \binom{b}{a} = (-1)^{\frac{a-1}{2}\frac{b-1}{2}} \binom{b \mod_{odd} a}{a}.$$
The odd modular reduction $\mod_{odd}$ consists to choose the odd value of $(b \mod a)$ in $\{-(a-1), \ldots, 0, \ldots, a-1\}$. The new expression involves two smaller integers. We can then compute $\binom{\Delta''}{p}$ by repeating the above process. This leads to a sequence of Euclidean division with odd remainders starting from $a_1 = p$ and $a_2 = \Delta''$ as follows:
$$\begin{aligned} a_1 &= q_1 a_2 + \varepsilon_3 a_3 \text{ with } \varepsilon_3 = \pm 1 \text{ and } 0 \le a_3 < a_2 \text{ and } a_3 \text{ odd},\\ a_2 &= q_2 a_3 + \varepsilon_4 a_4 \text{ with } \varepsilon_4 = \pm 1 \text{ and } 0 \le a_4 < a_3 \text{ and } a_4 \text{ odd},\\ \vdots &= \quad \vdots\\ a_{n-2} &= q_{n-1} a_{n-1} + \varepsilon_n a_n \text{ with } \varepsilon_3 = \pm 1 \text{ and } a_n = 1. \end{aligned}$$
The fact that $a_n = 1$ comes from $\gcd(\Delta', p) = 1$ since p is prime and that the above sequence of Euclidean division is a specific execution of the Euclidean algorithm. At the end we have:
$$\binom{\Delta'}{p} = \binom{2}{p}^t (-1)^{\frac{a_1-1}{2}\frac{a_2-1}{2}} \prod_{i=2}^{n-1} (-1)^{\frac{\varepsilon_i a_i - 1}{2}\frac{\varepsilon_i a_{i+1}-1}{2}}.$$

C. Computation of the square root

For the computation of the square root of an element $a \in \mathbb{F}_p$ we use the method of Shanks-Tonelli [26]. We assume that $p - 1 = q \times 2^t$ with t not too large. Let g be a generator of $\mathbb{F}_p^\star$ then $\gamma = g^q$ generates the subgroup of order 2^t of $\mathbb{F}_p^\star$. Let $a' = a^q$, then the approach of Shanks-Tonelli is based on the following expression of the square root of a:

$$\sqrt{a} = a^{(q+1)/2} \times (\sqrt{a'})^{-1}.$$

The above expression is correct since we have

$$(a^{(q+1)/2} \times (\sqrt{a'})^{-1})^2 = a^q \times a \times a'^{-1} = a$$

We can compute $a' = a^q$ and $a^{(q+1)/2}$ using a single exponentiation: we first compute $a^{(q-1)/2}$ and then we get

$$a' = (a^{(q-1)/2})^2 \times a \text{ and } a^{(q+1)/2} = a^{(q-1)/2} \times a.$$

To compute $\sqrt{a'}^{-1}$ we first compute the discrete logarithm e of a' in base γ the generator of the subgroup of order 2^t of $\mathbb{F}_p^\star$. To do this Shanks and Tonelli suggest to use the Pollig-Hellman approach to compute $e = \log_\gamma(a')$. This requires $(t-1)(t-2)/2$ squarings and t multiplications in $\mathbb{F}_p$. This method is shown in Algorithm 7. When we obtain $e = \log_\gamma(a')$ with this algorithm, we can get $\sqrt{a'}^{-1}$ by computing $(\gamma^{-1})^{e/2}$ assuming that γ^{-1} is precomputed and e i seven.

Algorithm 7 Computation of $\log_\gamma$

```
Require: $b$ a $2^t$-th root of unity and $\gamma$ a primitive $2^t$-th root of unity.
Ensure: $e = \log_\gamma(b)$
  Precomputation: $T[i] = \gamma^{-2^i}$ for $i = 1, \dots, t-1$.
  for $i = t-1$ to $0$ by $-1$ do
    $c \leftarrow b^{2^i}$
    if $c \neq 1$ then
      $e \leftarrow e + 2^{t-1-i}$
      $b \leftarrow b \times T[t-1-i]$
  return $e$
```

Complexity of Algorithm 7. The overall computation of a square root first requires an exponentiation $a^{(q-1)/2}$ which costs in average $\frac{\log_2(p)}{2}M + \log_2(p)S$. It also requires one execution of Algorithm 7 which has a cost of $tM + \frac{t(t+1)}{2}S$. The final operations are the computation $\sqrt{a'}^{-1}$ from e and γ^{-1} which has a complexity of $tS + tM$, and a final product $a^{(q-1)/2} \times \sqrt{a'}^{-1}$. The overall complexity is as follows:

$$(\frac{\log_2(p)}{2} + 2t + 1)M + (\log_2(p) + t(t+3)/2)S.$$

D. *Complexity of divisor splitting*

Let us now evaluate the complexity of a divisor splitting. We assume that a Jacobi symbol is computed with the exponentiation method which requires in average $\frac{\log_2(p)}{2}M + \log_2(p)S$. Since we execute in average one loop while iteration, we get the contribution of each step of Algorithm 6 and the total cost of the divisor splitting as shown in Table I. Not that we assume that a multiplication by small constant has the same cost as an addition in $\mathbb{F}_p$.

Table I
COMPLEXITY OF DIVISOR SPLITTING ON GENUS 2 HYPERELLIPTIC CURVES

Steps	Operations	Cost
(Step 3) + (Step 8)	Discriminant Δ'	$2S + 4a$
(Step 4) + (Step 9)	Jacobi symbol	$\log_2(p)M + 2\log_2(p)S$
(Step 6)	Divisor doubling	$1 DoublingAff$
(Step 7)	Division by 2	$1a$
(Step 10)	Square root (Step 10:)	$(\frac{\log_2(p)}{2} + 2t + 1)M + (\log_2(p) + t(t+3)/2)S$
(Step 11)	Roots x'_0, x'_1	$5a$
(Step 12) +(Step 13)	D'_0, D'_1	$2M + 4a$
Total cost		$(\frac{3\log_2(p)}{2} + 2t + 3)M + (3\log_2(p) + 2 + t(t+3)/2)S$ $+1\text{DoublingAff} + 14a$

E. *Addition formula in affine coordinates of a degree 2 with a degree 1 divisor*

In the considered approach for scalar multiplication (Algorithm 5) we need to perform at each loop an addition of a divisor of degree 2 with a degree 1 divisor. To render this approach efficient on the Jacobian of a genus 2 hyperelliptic curve $H(\mathbb{F}_p)$ we need efficient formula for this kind of divisor addition.

In [7] Costello and Lauter provide formulas for the addition and doubling of degree 2 divisors. Let us adapt their approach to provide, in affine coordinates, a formula for the addition of a degree 2 divisor with a degree 1 divisor.

In [7] Costello and Lauter consider the divisor composition and reduction as algebraic operations on the divisor coordinates instead of arithmetic of polynomials as it is done in Algorithm 1 and 2. We apply their method for the case of an addition of a degree 2 divisor $D = (u(X), v(X))$ and degree 1 divisor $D' = (u'(X), v'(X))$. For the composition of the two divisors we assume that we do not have to perform a semi-reduction which means that $\gcd(u, u') = 1$. In the composition of the two divisors we compute $l(X) = \sum_{i=0}^{3} l_i X^i$ satisfying

$$\begin{aligned} l(X) - v(X) &\equiv 0 \mod u(X), \\ l(X) - v'(X) &\equiv 0 \mod u'(X). \end{aligned}$$

In the above identities, if we explicit the expressions of $u(X), v(X), u'(X), v'(X)$ and $l(X)$, and then reduce the resulting expression modulo $u(X)$ and $u'(X)$, we get the following identities

$$\begin{aligned}(l_1 - l_2u_1 - v_1)X + (l_0 - l_2u_0 - v_0) &= 0,\\ l_2u_0'^2 - u_0'l_1 + l_0 - v_0' &= 0.\end{aligned}$$

These two polynomial identities can be set in a 3×3 linear system

$$\begin{bmatrix} 1 & 0 & -u_0 \\ 0 & 1 & -u_1 \\ 1 & -u_0' & u_0'^2 \end{bmatrix} \begin{bmatrix} l_0 \\ l_1 \\ l_2 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_0' \end{bmatrix}$$

Subtracting the first row to the last one in the above system provides the following 2×2 linear system:

$$\begin{bmatrix} 1 & -u_1 \\ -u_0' & u_0'^2 + u_0 \end{bmatrix} \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_0' - v_0 \end{bmatrix}$$

We solve this system using Cramer's method and this leads to the following expression for l_2, l_1 and l_0:

$$l_2 = \frac{u_0'v_1 + v_0' - v_0}{u_0'^2 + u_0 - u_1u_0'} \text{ and } \left\{ \begin{aligned} l_1 &= v_1 + u_1l_2 \\ l_0 &= v_0 + u_0l_2 \end{aligned} \right. \tag{1}$$

The next step is the reduction of the divisor $(u(X) \times u'(X), l(X))$. To reduce it we first compute $u''(X) = \frac{l(X)^2 - f(X)}{u(X)u'(X)}$: the coefficients of $u''(X)$ are as follows

$$\begin{aligned} u_1'' &= -l_2^2 - (u_1 + u_0') \\ u_0'' &= -(u_0 + u_1u_0' + u_1''(u_1 + u_0') - 2l_1l_2 + f_3 \end{aligned} \tag{2}$$

Finally we compute the coefficient v_0'' and v_1'' from $v''(X) = -l(X) \mod u''(X)$ using a direct approach. The resulting formula is given in Algorithm 8. Its complexity is equal to:

$$10M + S + I + 18a.$$

Algorithm 8 `Add21Aff`$(u_1, u_0, v_1, v_0, u_0', v_0')$

```
$l_2 \leftarrow (v_0' - v_0 + u_0' \times v_1)/((u_0' - u_1) \times u_0' + u_0)$        $//3M + I + 4a$
$l_1 \leftarrow v_1 + u_1 \times l_2$        $//M + a$
$l_0 \leftarrow v_0 + u_0 \times l_2$        $//M + a$
$u_1'' \leftarrow -l_2^2 - (u_1 + u_0')$        $//S + 3a$
$u_0'' \leftarrow -(u_0 + u_1 \times u_0' + u_1'' \times (u_1 + u_0)) - 2l_1 \times l_2 + f_3$        $//3M + 7a$
$v_1'' \leftarrow (l_2 \times u_1'' - l_1)$        $//M + a$
$v_0'' \leftarrow (l_2 \times u_0'' - l_0)$        $//M + a$
return $u_1'', u_0'', v_1'', v_0''$        // Total $= 10M + S + I + 18a$
```

F. Inversion free formula for addition of a degree 2 with a degree 1 divisor

Tanja Lange introduced in [19] an inversion free formula by using a projective form of the Mumford representation. In this coordinate system a degree two divisor D is given by five field elements

$$(U_1, U_0, V_1, V_0, Z) \text{ such that } D = (X^2 + (U_1/Z)X + U_0/Z, (V_1/Z)X + V_0/Z).$$

Tanja Lange took advantage of this system of coordinates by replacing a costly field inversion in the doubling and addition formulas, with a number of multiplications and additions in $\mathbb{F}_p$, leading to a more efficient formula. The formulas of Lange for addition and doubling was later slightly improved by Costello and Lauter in [7].

Recently, Costello and Hisil in [12] considered a new set of projective coordinates, the following ZW Jacobian coordinates:

$$(U_1, U_0, V_1, V_0, Z, W) \text{ such that } D = (X^2 + (U_1/Z^2)X + U_0/Z^4, (V_1/(Z^3W))X + V_0/(Z^5W)).$$

Using these projective coordinates they could modify the formula of Costello-Lauter [7] and reduce the number of field operations. Moreover their addition formula take as input two divisors D_1 and D_2 with the same value for Z and W. This kind of formula is called co-ZW formula (co-Z formula was originally introduced in [21] for Jacobian coordinates over elliptic curve).

The doubling and addition formulas of Costello and Hisil are the most efficient inversion free formula of degree two divisors in $Jac(H(\mathbb{F}_p))$. In order to use their formulas in our regular scalar multiplication (Algorithm 5) we need an addition formula in the ZW Jacobian coordinate system of a degree 2 with a degree 1 divisor as inputs. Specifically we use the following coordinate system for a degree 1 divisor

$$(U_0, V_0, Z, W) \text{ such that } D = (X + \tfrac{U_0}{Z^2}, \tfrac{V_0}{Z^5W})$$

We then modify the affine formula of Algorithm 8 in order to obtain a co-ZW formula for addition of a degree 2 divisor with a degree 1 divisor. Injecting the ZW Jacobian coordinates of $D = (U_0, U_1, V_0, V_1, Z, W)$ and $D' = (U_0', V_0', Z, W)$ in Algorithm 8, and arranging the resulting formula we could get the following expressions for the computation of the coordinates $(U_0'', U_1'', V_0'', V_1'', Z'', W'')$ of $D'' = D + D'$:

$$\begin{array}{rcl}
A &=& (V_0' - V_0 + U_0'V_1), \\
B &=& (U_0' - U_1)U_0' + U_0, \\
L_1 &=& (BV_1(BW)^2 + U_1A(BW)^2), \\
Z'' &=& BZW, \\
W'' &=& 1, \\
U_1'' &=& A^2 - (U_1 + U_0')BW, \\
U_0'' &=& -U_0(BW)^4 - ((U_1(BW)^2)U_0'(BW)^2) + (BW)^4 f_3 Z^4 - U_1''(U_1(BW)^2 + U_0'(BW)^2), \\
 && -2L_1(BW)^2A, \\
V_1'' &=& (AU_1'' - L_1), \\
V_0'' &=& (A(U_0'' - U_0(BW)^2) - V_0B^4W^5).
\end{array} \tag{3}$$

For completeness we provide a quick proof of the validity that the above expressions (3) lead to the correct ZW Jacobian coordinates of $D'' = D + D'$ in the appendix. We provide in Algorithm 9 a method to compute the above expressions (3) step by step. We also provide the complexity of each step and the overall complexity for Add21Proj_coZW.

Algorithm 9 Add21Proj_coZW($U_1, U_0, V_1, V_0, Z, W, U_0', V_0'$)

```
$A \leftarrow (V_0' - V_0 + U_0' \times V_1)$                                          // 1M +2a
$B \leftarrow (U_0' - U_1) \times U_0' + U_0$                                        // 1M +2a
$BW \leftarrow B \times W$                                                           // 1M
$BW2 \leftarrow (BW)^2$                                                              // 1S
$NU_1 \leftarrow BW2 \times U_1$                                                     // 1M
$NU_0' \leftarrow BW2 \times U_0'$                                                   // 1M
$temp \leftarrow NU_1 + NU_0'$                                                       // 1a
$U_1'' \leftarrow -A^2 - temp$                                                       // S+2a
$BW4 \leftarrow BW2^2$                                                               // S
$NU0 \leftarrow BW4 \times U_0$                                                      // 1M
$NV1 \leftarrow B \times V_1 \times BW2$                                             // 2M
$L1BW2 \leftarrow (NV1 + NU1 \times A)$                                              // 1M+a
$U_0'' \leftarrow -NU_0 - (NU_1 \times NU_0') + BW4 \times f_3 \times Z^4 - U_1'' \times temp - 2(L1BW2) \times A$   // 4M+2S+D+5a
$V_1'' \leftarrow A \times U_1'' - L1BW2$                                            // 1M+a
$NV_0 \leftarrow V_0 \times B \times BW4$                                            // 2M
$V_0'' \leftarrow A \times (U_0'' - NU_0) - NV_0$                                    // 1M+2a
$Z'' \leftarrow BW \times Z$                                                         // 1M
return $U_1'', U_0'', V_1'', V_0'', Z'', 1, NU_1, NU_0, NV_1, NV_0$                  // Total = 18M+D+5S+16a
```

The formula Add21Proj_coZW outputs in addition to $D'' = D + D'$ the four coordinates NU_1, NU_0, NV_1, NV_0. These coordinates are the ones of D such that D and D'' are co-ZW. In other words

$$(NU_1/Z''^2, NU_0/Z''^4, NV_1/Z''^3, NV_0/Z''^5) = (U_1/Z^2, U_0/Z^4, V_1/(Z^3W), V/(Z^5W)).$$

This can be useful when we have to re-add D to D'' since then we can use the co-ZW addition formula of [12].

G. Complexity results and comparison

In this subsection we analyze and compare the complexities of divisor operations and the regular scalar multiplication algorithms on $Jac(H(\mathbb{F}_p))$.

- *Addition and doubling complexity comparison.* In Table II we review the complexity of the formulas of the literature for the divisor operations on $Jac(H(\mathbb{F}_p))$. We also provide the complexity of the proposed formulas for Add21. For doubling and addition in ZW Jacobian coordinates the best formulas are due to Costello and Hisil [12]. Their formula Add22_CoZW requires that the two inputs have the same Z and W coordinates. For the affine coordinates the best approaches for Add22 and Doubling2 are from [7]. If we consider Add21 in affine coordinates the best approach before this work was in [19]. We can see that the proposed formula of Algorithm 8 improves the approach of [19] by 6 additions. In projective coordinates we could not find any formula in the literature to add a degree 2 with a degree 1 divisor. If we compare our formula Add21Proj_ZW (Algorithm 9) with the full addition of divisor Add22Proj_coZW: we notice that the number of multiplications/squarings is reduced by 5 and the number of additions by 6.

Table II
COST OF THE OPERATIONS ON THE JACOBIAN OF A GENUS 2 HYPERELLIPTIC CURVE

Operation	Coordinate Syst.	Cost
Add22Aff [7]	Affine	$17M + 4S + I + 56a$
Doubling2Aff [7]	Affine	$19M + 6S + I + 54a$
Add21Aff [18]	Affine	$10M + 1S + I + 24a$
Proposed Add21Aff (Algorithm 8)	Affine	$10M + S + I + 18a$
Doubling2Proj [12]	ZW Jac. Proj.	$26M + 8S + 2D + 25a$
mAdd22Proj [12]	ZW Jac. Proj.	$32M + 5S + 22a$
Add22Proj [12]	ZW Jac. Proj.	$41M + 7S + 22a$
Add22Proj_coZW [12]	ZW Jac. Proj.	$25M + 3S + 22a$
mDoubleAdd22 [12]	ZW Jac. Proj.	$57M + 8S + 42a$
Proposed Add21Proj_CoZW (Algorithm 9)	ZW Jac Proj.	$18M + D + 5S + 16a$
Convert2_AfftoZW (Degree 2)	-	$7M + 2S$
Convert2_ZWtoAff (Degree 2)	-	$8M + 2S + I$
Convert1_AfftoZW (Degree 1)	-	$4M + 2S$

In the sequel we will need to convert a divisor given in affine coordinate into ZW coordinate with a given Z and W in order to apply ADD22Proj_coZW or ADD21Proj_coZW. These conversions are detailed in the appendix, their complexities are shown in Table II.

- *Regular scalar multiplication complexity.*
  Using the complexities reported in Table II for doubling and addition on $Jac(H(\mathbb{F}_p))$ we can evaluate the complexity of the regular exponentiation algorithms reviewed in Subsection II-B. We consider a scalar K of bit length ℓ. We consider the following approaches:
  - *Double-and-add-always in affine coordinates.* This consists of ℓ Doub2_Aff and Add2_Aff , leading to a complexity of $\ell(36M + 10S + 2I + 110a)$.
  - *Regular multiplication based on divisor splitting in affine coordinates.* This consists of ℓ Doubling2Aff and Add21Aff plus the cost of a divisor splitting (cf. Table I). The overall complexity is then equal to $\ell(29M + 7S + 2I + 71a) + (3\log_2(p)/2 + 2t + 25)M + (3\log_2(p) + t(t+3)/2 + 9)S + I + 71a$.
  - *Double-and-add-always in ZW Jacobian coordinates.* In this case Algorithm 3 consists in a sequence of a doubling and an addition of D with either R_0 or R_1. We take advantage of Add22Proj_coZW by rewritting the loop iteration of Algorithm 3 as follows:

    $R_1 \leftarrow$ Doubling2Proj(R_1)

    $ND \leftarrow$ Convert2_AfftoZW$(D, Z_{R_{k_i}}, W_{R_{k_i}})$ // ND is the same divisor as D but co-ZW with R_{k_i}

$R_{k_i} \leftarrow$ ADD22_ZW(R_{k_i}, ND)

We add the cost of Doubling2Proj, Convert2_AfftoZW and ADD22_ZW to get the cost of a loop iteration. We add the final conversion of R_1 to affine coordinates and we obtain the following complexity:

$$\ell(58M + 13S + 2D + 47a) + 8M + I + 2S.$$

– *Signed-double-and-add-always in ZW Jacobian coordinates.* Again, we take advantage of the efficiency of ADD22Proj_coZW by rewritting the loop iteration as follows:

$ND_{k_i} \leftarrow$ Convert2_AfftoZW(D_{k_i}, Z_R, W_R) // Z_R and W_R are the coordinates of R

$R2, NR \leftarrow$ ADD22_ZW(R, ND_{k_i}) // NR is the same divisor as R but co-ZW with $R2$

$R \leftarrow$ ADD22_ZW$(R2, NR)$

This requires two ADD22_ZW and one Convert2_AfftoZW per loop iteration. We get the following overall complexity

$$\ell(57M + 8S + 44a) + 8M + I + 2S.$$

– *Regular multiplication based on divisor splitting in ZW Jacobian coordinates.* We have to compute at each loop iteration $2R + D_{k_i}$ where D_{k_i} is a degree 1 divisor. We use the following strategy for the loop iteration in Algorithm 5:

$ND_{k_i} \leftarrow$ Convert1_AfftoZW(D_{k_i}, Z_R, W_R) // Z_R and W_R are the coordinates as R

$R2, NR \leftarrow$ ADD21_ZW(R, ND_{k_i}) // NR is the same divisor as R but co-ZW with $R2$

$R \leftarrow ADD22\_ZW(R2, NR)$

Consequently, this requires one ADD21_ZW, one ADD22_ZW and one Convert2_AfftoZW per loop iteration. The overall complexity includes the cost of the splitting and the final conversion to affine coordinates and is equal to

$$\ell(47M + D + 10S + 38a) + (3\log_2(p)/2 + 2t + 33)M + (3\log_2(p) + t(t+3)/2 + 11)S + 2I + 71a$$

In Table III we report the cost of the above approaches, for the proposed approach we assume that the key length $\ell = 2\log_2(p)$ in order to have a better understanding of the asymptotic complexity. We can notice that for both cases, affine and ZW Jacobian coordinates, our approach leads to a better complexity than the signed and unsigned Double-and-add-always approaches. For the affine case the improvement is in the number of multiplications and squarings, so if the cost of an inversion is important compared to a multiplication probably we will not see a huge impact of this improvement. For the case of ZW Jacobian coordinates we save roughly 4 M/S over 65 M/S per loop iteration, this provides an improvement around 6%.

H. Implementation results

We implemented all the scalar multiplication apporaches mentioned in the previous subsection in order to have practical evaluation of the speed-up provided by the proposed divisor splitting approach. We used the following strategies in our implementations:

Table III

COMPLEXITY OF REGULAR SCALAR MULTIPLICATION ON GENUS 2 CURVES

Algorithm	Coordinates	Cost
Double-and-add-always	Affine	$\ell(36M+10S+2I+110a)$
Proposed regular algorithm	Affine	$\ell(30.75M+8.5S+2I+71a)+(2t+25)M+(t(t+3)/2+9)S+I+71a$
Double-and-add-always	ZW Jac.	$\ell(58M+13S+2D+47a)+8M+I+2S$
Signed-double-and-add-always	ZW Jac.	$\ell(57M+8S+44a)+8M+I+2S$
Proposed regular algorithm	ZW Jac.	$\ell(48.75M+D+11.5S+38a)+(2t+33)M+(t(t+3)/2+11)S+2I+71a$

- Our code was written in C langage, compiled with gcc and run on the three platform Intel Core i5-3210, Intel Xeon E5-2650v4 and ARMv7.
- We considered three hyperelliptic curves defined over $\mathbb{F}_p$ for $p = 2^{94}-3$, $p = 2^{122}-3$ and $p = 2^{300}-3$, respectively. Their group size corresponds to a security level of $188, 244$ and 300.
- For field multiplication, addition and subtraction we used the strategy of Langley [20]. We store a field element in a number of computer words keeping in each word a number of spared bits to handle carries. Integer addition and multiplication are done using schoolbook approaches.
- For the inversion we implemented one approach based on the little Fermat theorem [2] and one approach using Euclidean algorithm [2] (in this later case we used the low level function of gmp [1]). We selected the Euclidean algorithm in our scalar multiplication code since it appeared to be the fastest among these approaches.
- For the computation of the Jacobi symbol we implemented the two approaches reviewed in Subsection III-B: the first one is based on an exponentiation in $\mathbb{F}_p$ and the second one uses the Euclidean-like algorithm of Eisenstein [28]. Both were not fully optimized but the approach based on the exponentiation showed the best performances.

The timing results for the scalar multiplication are shown in Table IV. As we can see for all cases the approach using projective coordinates are always better that the one using affine coordinates. Moreover, these timings indicate that in practice the proposed approach is still efficient compared to the best approaches of the literature. In affine the improvement is around 1% to 4% on Intel processor and 0%-8% on the ARM processor, this is explained by the dominant time spent in the large number of inversions which is the same for both approaches. The proposed approach reduces the computation time by 4% to 10% in projective coordinates. This is in the range of what we expected based on the complexity analysis.

IV. REGULAR SCALAR MULTIPLICATION OVER A GENUS 3 HYPERELLIPTIC CURVE

In this section we extend the strategy presented in the Section III to the case hyperelliptic curves of genus three. Over $\mathbb{F}_p$ such curves are given by an equation of the following form

$$Y^2 = X^7 + f_5X^5 + f_4X^4 + f_3X^3 + f_2X^2 + f_1X + f_0.$$

Table IV

TIMINGS OF REGULAR SCALAR MULTIPLICATION ON GENUS 2 HYPERELLIPTIC CURVES

Method	Coord.	Security level	Timing (10^3 clocks-cycles)		
			Intel Core i5-3210M	Intel Xeon E5-2650v4	ARMv7
Double-and-add-always	Affine	192	2523	931	3002
Proposed	Affine	192	2493	922	3009
Signed-double-and-add-always	ZW Jac.	192	586	187	1240
Proposed	ZW Jac.	192	537	180	1207
Double-and-add-always	Affine	244	3965	1455	5656
Proposed	Affine	244	3927	1450	5358
Signed-double-and-add-always	ZW Jac.	244	760	257	2671
Proposed	ZW Jac.	244	706	232	2467
Double-and-add-always	Affine	300	7289	2570	9212
Proposed	Affine	300	6972	2499	8406
Signed-double-and-add-always	ZW Jac.	300	1779	543	4517
Proposed	ZW Jac.	300	1655	528	4174

In order to render Algorithm 5 efficient over genus 3 hyperelliptic curves we need a method to split the base divisor and an addition formula adapted to this splitting. It is possible to split the base divisor D as $D = D_1 - D_0$ where D_1 is a degree 2 divisor and D_0 is a degree 1 divisor. But in this case Algorithm 5 would not be regular since an addition with a degree 2 divisor is more costly than an addition with a degree one divisor. Our first goal is to split a divisor of degree 3 as a difference of two degree 2 divisors in order to keep Algorithm 5 fully regular. Afterwards we will provide Add32 formulas for the addition of a degree 3 divisor with a degree 2 divisor.

A. Divisor splitting over a genus 3 hyperelliptic curve

We are going to split a divisor D of degree 3 as a difference of two degree 2 divisors. Beforehand, recall that a degree 3 divisor is $D = (P_0) + (P_1) + (P_2) - 3(\infty)$ and let $D = (u(X), v(X))$ be its Mumford representation. The roots x_i of $u(X)$ are the abscissa of P_i and the polynomial $v(X)$ is such that $v(x_i) = y_i$ is the ordinate of P_i for $i = 0, 1, 2$. The splitting of a divisor D into two degree 2 divisors is done as follows:

1) We first compute $D' = D + D_0$ where D_0 is a fixed degree 1 divisor.
2) Let $D' = (u'(X), v'(X))$, we try to find a root $\alpha \in \mathbb{F}_p$ of $u'(X)$ if such root does not exist we double D and go to 1).
3) We compute $u_1(X) = u'(X)/(X - \alpha)$ and $v_1 = v(X) \mod u_1(X)$ and we set $D_1 = (u_1, v_1)$.
4) We consider $\beta = v'(\alpha)$ and the divisor $D'' = (X - \alpha, \beta)$. We have the splitting

$$D' = D_1 + D'' \tag{4}$$

as a sum of a degree 2 and a degree 1 divisor.

5) We compute $D_2 = D_0 - D''$ and we output (D_1, D_2).

We can check that D_1, D_2 is the required splitting of D as a difference of two degree 2 divisors. Indeed we have

$$\begin{aligned} D &= D' - D_0 \quad \text{(from 1))} \\ &= D_1 + D'' - D_0 \quad \text{(with (4))} \\ &= D_1 - D_2 \quad \text{(from 5)).} \end{aligned}$$

Consequently to get the splitting of D we have to compute a root of a degree 3 polynomial. We use the classical formula due to Cardan to compute the root of a degree 3 polynomial. Let $u'(X) = X^3 + u'_2X^2 + u'_1X + u'_0$ be a degree 3 polynomial, then Cardan's formula computes a root α of $u'(X)$ as follows:

$$\begin{cases} r &= (u'_1 - \frac{u'^2_2}{3}), \\ s &= (u'_0 - \frac{u'^3_2}{27}), \\ \Delta &= 4r^3 + 27s^2, \\ \Omega &= -\frac{s}{2} - \frac{1}{2}\sqrt{\frac{\Delta}{27}}, \\ \alpha &= -u'_2 + \sqrt[3]{\Omega} + \frac{-r}{3\sqrt[3]{\Omega}}. \end{cases}$$

In the above formula we notice that the most costly operations are the square root computation $\sqrt{\frac{\Delta}{27}}$ and the cube root computation $\sqrt[3]{\Omega}$. In the previous section we have already seen how to compute a square root in $\mathbb{F}_p$. For a cube root we consider the following cases:

- If $\gcd(p-1, 3) = 1$ then in this case for each element $a \in \mathbb{F}_p$ there always exists a unique cube root in $\mathbb{F}_p$. If we set $q = 3^{-1} \mod (p-1)$ then the cube root of a is $b = a^q \mod p$.
- If $p - 1 = 3q$ with $\gcd(q, 3) = 1$ then we have first to decide if a is cube in $\mathbb{F}_p$, this is the case if $a^q = a^{(p-1)/3} = 1$. If this is the case we can compute the cube root of a as follows
  - If $q \equiv 1 \mod 3$. Then $b = a^{-\frac{q-1}{3}}$ satisfies $b^3 = a^{-q} \times a = a$.
  - If $q \equiv 2 \mod 3$. Then $b = a^{\frac{q+1}{3}}$ satisfies $b^3 = a^q \times a = a$.
- For other cases, i.e., p such that $\gcd(\frac{p-1}{3}, 3) \neq 1$, we can adapt the method of Shanks-Tonelli (Subsection III-C) to the case of cube-root computation. This extension is straightforward so we do not give the details here.

The conclusion of the above discussion is that a cube-root can be computed through an exponentiation a^e in $\mathbb{F}_p$ were e is of bit-length $\cong \log_2(p)$.

The splitting of a degree 3 divisor into two degree 2 divisors using the proposed strategy it is shown in Algorithm 10.

B. Complexity of a divisor splitting

In order to evaluate the complexity of a divisor splitting (Algorithm 10), we need to know the average number of time we execute the loop iteration of the algorithm. This will give us the average number of time we have to determine if $\Delta/27$ is a square and if Ω is a cube and compute their respective square and cube root. The number of

Algorithm 10 Divisor splitting on genus 3 curve

```
Require: A degree 3 divisor $D = (u(X), v(X))$ of a Jacobian of a genus 3 hyperelliptic curve of odd order $N$, a
    scalar $K$ and a fixed degree 1 divisor $D_0 = (X - x_0, y_0)$
Ensure: $K'$ and $\widetilde{D} = D_1 - D_2$ with $D_1$ and $D_2$ of degree 2 and $K' \cdot \widetilde{D} = K \cdot D$.
 1: $\widetilde{D} \leftarrow D, K' \leftarrow K, \tau \leftarrow -1$
 2: $D' = (u'(X), v'(X)) \leftarrow Add(\widetilde{D}, D_0)$
 3: $r \leftarrow u'_1 - \frac{u_2'^2}{3}, s \leftarrow (u'_0 - \frac{u_2'^3}{27}), \Delta' \leftarrow \frac{4r^3}{27} + s^2$
 4: $\sigma \leftarrow JacobiSymbol(\Delta')$
 5: if $\sigma = 1$ then
 6:     $\Omega \leftarrow -\frac{s}{2} - \frac{1}{2}\sqrt{\Delta'}, \tau \leftarrow IsCube(\Omega)$
 7: while $\tau \neq 1$ do
 8:     $K' = K' \times 2^{-1} \mod N$
 9:     $\widetilde{D} \leftarrow Doubling(\widetilde{D})$
10:     $D' = (u'(X), v'(X)) \leftarrow Add(\widetilde{D}, D_0)$
11:     $r \leftarrow u'_1 - \frac{u_2'^2}{3}, s \leftarrow (u'_0 - \frac{u_2'^3}{27}), \Delta' \leftarrow \frac{4r^3}{27} + s^2$
12:     $\sigma \leftarrow JacobiSymbol(\Delta')$
13:     if $\sigma = 1$ then
14:         $\Omega \leftarrow -\frac{s}{2} - \frac{1}{2}\sqrt{\Delta'}, \tau \leftarrow IsCube(\Omega)$
15: $\alpha \leftarrow -u'_2 + \sqrt[3]{\Omega} + \frac{-r}{3\sqrt[3]{\Omega}}$
16: $D_1 \leftarrow (u'(X)/(X - \alpha), v'(X) \mod (u'(X)/(X - \alpha)))$
17: $D_2 \leftarrow Add(D_0, (X - \alpha, -v'(\alpha)))$
18: return $K', \widetilde{D}, D_1, D_2$
```

time we stay in the while loop is related to the probability that Δ is a square and Ω is a cube. These probabilities are established in the following lemma.

Lemma 3. *Let Δ and Ω be two elements in $\mathbb{F}_p$ with p an odd large prime.*

- *The probability that Δ is a square is $\frac{1}{2} + \frac{1}{p} \cong \frac{1}{2}$*
- *If $3 \not| (p-1)$ the probability that Ω is a cube is 1.*
- *If $3 \,|\, (p-1)$ the probability that Ω is a cube is $\frac{1}{3} + \frac{2}{3p} \cong \frac{1}{3}$.*

Using these probabilities we can estimate the number of time we stay in the while loop in Algorithm 10. Let $\mathfrak{N}$ be the random variable equal to the number of time we execute the loop iteration in Algorithm 10. For the sake of simplicity we only consider the case $3|(p-1)$. We first evaluate probability that the computation of a root of

$u(X)$ fails. This probability is as follows:

$$\begin{aligned}\mathbb{P}(\text{u(X) has not a root})) &= \mathbb{P}(\{\tfrac{\Delta}{27} \text{ is not a square }\} \cup \{\tfrac{\Delta}{27} \text{ is a square and } \Omega \text{ is not a cube }\}) \\ &= \tfrac{1}{2} + \tfrac{1}{2} \times \tfrac{2}{3} = \tfrac{5}{6}\end{aligned}$$

We then obtain the probability that we get a root of $u'(X)$ after $\mathfrak{N} = i$ iterations, we fail i times to find a root and then we succeed.

$$\mathbb{P}(\mathfrak{N} = i) = \frac{5}{6}^i \times \frac{1}{6}.$$

Then the expected number of iterations is

$$\mathbb{E}[\mathfrak{N}] = \textstyle\sum_{i=1}^{\infty} i \times \frac{5}{6}^i \times \frac{1}{6} = \frac{5}{36} \times \frac{1}{(1-\frac{5}{6})^2} = 5$$

Now we can derive the complexity of Algorithm 10. We assume that a Jacobi-symbol and a square-root can be computed through a single exponentiation. This is also the case for IsCube and cube root computation. We also assume that the cost of an exponentiation in $\mathbb{F}_p$ is $\log_2(p)S + \frac{\log_2(p)}{2}M$ in average. Based on this facts, the complexity of Algorithm 10 is evaluated step by step in the Table V.

Table V

COMPLEXITY ANALYSIS OF DIVISOR SPLITTING ON GENUS 3 HYPERELLIPTIC CURVE

Step	Cost
(Step 2) $+5\times$ (Step 10)	6Add31Aff
(Step 3) $+5\times$ (Step 11)	$24M + 18S + 30a$
(Step 4) $+5\times$ (Step 12)	$3\log_2(p)M + 6\log_2(p)S$
$\frac{1}{2} \times$ ((Step 6) $+ 5 \times$ (Step 14))	$\frac{3\log_2(p)}{2}M + 3\log_2(p)S + 9a$
$5\times$ (Step 8)	$5a$
$5\times$ (Step 9)	5 DoublingAff
(Step 15)	$M + 3a + I$
(Step 16)	$3M + 5a$
(Step 17)	Add11$+M + 2a$
Total cost	6Add31 + Add11 + $(\frac{9\log_2(p)}{2} + 29)M$ $+(9\log_2(p) + 18)S + 54a$

C. Addition formula of degre 3 and degree 2 divisors

In the proposed regular scalar multiplication (Algorithm 5) on the Jacobian of a genus 3 hyperelliptic curve we have to perform a sequence of doublings of degree 3 divisors followed by an addition with degree 2 divisor. For the doubling formula in affine and projective coordinates we use the formula provided in [9]. For the addition of a degree 3 divisor with a degree 2 divisor (Add32), we provide a new formula using the approach of Costello and Lauter [7].

Let $D = (u(X), v(X))$ be a degree 3 divisor and $D' = (u'(X), v'(X))$ be a degree 2 divisor. We assume that $\gcd(u(X), u'(X)) = 1$ which means that if a point $P \in H(\mathbb{F}_p)$ is in D then neither P, neither $\bar{P}$ are in D'.

The method of Costello and Lauter first computes the composition of D and D' (cf. Algorithm 1). Indeed the Mumford representation of the unreduced divisor $D + D'$ is given by the product $u(X)u'(X)$ for the abscissas and, for the ordinates, $l(X)$ which satisfies $l(X) \equiv v(X) \mod u(X)$ and $l(X) \equiv v'(X) \mod u'(X)$. To get an explicit expression of the coefficients of $l(X)$ we express the reduction $l(X) \mod u(X)$ in terms of the coefficients of $l(X)$ and $u(X)$:

$$l(X) \mod u(X) \equiv (l_2 + (u_2^2 - u_1)l_4 - u_2 l_3)x^2 + (l_1 + (u_1u_2 - u_0)l_4 - u_1l_3)x^1 + l_0 - u_0l_3 + u_0u_2l_4)$$

Similarly the reduction of $l(X)$ modulo $u' = X^2 + u_1'X + u_0'$ gives the following

$$l(X) \mod u'(X) = (l_1 - u_1'l_2 + (2u_1'u_0' - {u_1'}^3)l_4 + ({u_1'}^2 - u_0')l_3))x^1 + (l_0 - u_0'l_2 + ({u_0'}^2 - u_0'{u_1'}^2)l_4 + u_0'u_1'l_3).$$

With the above two identities and using $v(X) = l(X) \mod u(X)$ and $v'(X) = \ell(X) \mod u'(X)$ we obtain the following linear system for $l_0, \ldots, l_4$

$$\begin{bmatrix} 1 & 0 & 0 & -u_0 & +u_0u_2 \\ 0 & 1 & 0 & -u_1 & (u_1u_2 - u_0) \\ 0 & 0 & 1 & -u_2 & (u_2^2 - u_1) \\ 1 & 0 & -u_0' & u_0'u_1' & ({u_0'}^2 - u_0'{u_1'}^2) \\ 0 & 1 & -u_1' & ({u_1'}^2 - u_0') & (2u_1'u_0' - {u_1'}^3) \end{bmatrix} \cdot \begin{bmatrix} l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_0' \\ v_1' \end{bmatrix} \tag{5}$$

Now, we arrange the above system by subtracting the first two rows to the last two rows, this leads to a 3×3 linear system for l_2, l_3 and l_4. We then add the first row of this 3×3 system multiplied by u_0' and u_1' to the two last rows. We obtain the following 2×2 linear system:

$$\begin{bmatrix} u_0'u_1' + u_0 - u_2u_0' & ({u_0'}^2 - u_0'{u_1'}^2 - u_0u_2 + (u_2^2 - u_1)u_0') \\ ({u_1'}^2 - u_0' + u_1 - u_2u_1') & (2u_1'u_0' - {u_1'}^3 - u_1u_2 + u_0 + (u_2^2 - u_1)u_1') \end{bmatrix} \cdot \begin{bmatrix} l_3 \\ l_4 \end{bmatrix} = \begin{bmatrix} v_0' - v_0 + u_0'v_2 \\ v_1' - v_1 + u_1'v_2 \end{bmatrix} \tag{6}$$

We set

$$\begin{aligned} a &= u_0'u_1' + u_0 - u_2u_0', & b &= ({u_0'}^2 - u_0'{u_1'}^2 - u_0u_2 + (u_2^2 - u_1)u_0'), \\ c &= ({u_1'}^2 - u_0' + u_1 - u_2u_1'), & d &= (2u_1'u_0' - {u_1'}^3 - u_1u_2 + u_0 + (u_2^2 - u_1)u_1'), \\ \Delta &= ad - bc. \end{aligned}$$

We apply Kramer's formula to get the solution for l_3 and l_4 of (6), the expressions for l_0, l_1 and l_2 follows from (5):

$$\left\{ \begin{aligned} l_3 &= \tfrac{1}{\Delta}\left(d \times (v_0' - v_0 + u_0'v_2) - b \times (v_1' - v_1 + u_1'v_1)\right), \\ l_4 &= \tfrac{1}{\Delta}\left(-c \times (v_0' - v_0 + u_0'v_2) + a \times (v_1' - v_1 + u_1'v_2)\right), \end{aligned} \right. \quad \text{and} \quad \left\{ \begin{aligned} l_0 &= u_0l_3 + u_0u_2l_4 + v_0, \\ l_1 &= u_1l_3 + (u_0 - u_1u_2)l_4 + v_1, \\ l_2 &= u_2l_3 + (u_1 - u_2^2)l_4 + v_2. \end{aligned} \right.$$

We now proceed to the reduction of the unreduced divisor $D'' = D + D' = (u(X)u'(X), l(X))$. We apply one iteration of the Cantor reduction algorithm (Algorithm 2): we have to compute $u''(X) = (l(X)^2 - f(X))/(u(X)u'(X))$. We use the following identity

$$(X^2 + u_1'X + u_0')(X^3 + u_2X^2 + u_1x + u_0)(X^3 + u_2''X^2 + u_1''X + u_0'') = \frac{(\ell_4X^4 + \ell_3X^3 + \ell_2X^2 + \ell_1X + \ell_0)^2 - f(X)}{\ell_4^2}$$

where $f(X) = X^7 + f_5X^5 + f_4X^4 + f_3X^3 + f_2X^2 + f_1X + f_0$. If we consider, in this equation, the four terms with the largest degree we get the following three coefficients of $u''(X)$

$$\begin{aligned} u''_2 &= (2\ell_4\ell_3 - 1)/\ell_4^2 - u'_1 - u_2, \\ u''_1 &= (2\ell_4\ell_2 + \ell_3^2)/\ell_4^2 - (u'_0 + u'_1u_2 + u_1) - u''_2(u'_1 + u_2), \\ u''_0 &= \tfrac{2\ell_3\ell_2 + 2\ell_4\ell_1 - f_5}{\ell_4^2} - ((u'_0u_2 + u'_1u_1 + u_0) + (u'_0 + u_1 + u'_1u_2)u''_2 + (u'_1 + u_2)u''_1). \end{aligned}$$

We then compute the coefficients of $v''(X)$ based on $v''(X) = -l(X) \mod u''(X)$ and this gives:

$$\begin{aligned} v''_0 &= -l_0 + u''_0l_3 - u''_0u''_2l_4, \\ v''_1 &= -l_1 + (u''_0 - u'_1u''_2)l_4 + u''_1l_3, \\ v''_2 &= -l_2 + (u''_1 - u''^2_2)l_4 + u''_2l_3. \end{aligned}$$

At the end we obtain the reduced Mumford representation $(u''(X), v''(X))$ of $D'' = D + D'$. We give a step by step formula for this computation in Algorithm 11 of the appendix.

Inversion free formula. If the inversion is too costly compared to a multiplication in the underlying field, it might be interesting to have an inversion free formula for Add32. In the literature Fan and Gong provides in [9] a set of formulas (for doubling, addition and mixed addition) for the following projective coordinates

$$(U_0, U_1, U_2, V_0, V_1, V_2, Z) \sim (U_0/Z, U_1/Z, U_2/Z, V_0/Z, V_1/Z, V_2/Z). \tag{7}$$

There is no formula in the literature with weighted coordinates like in [12], and it seems to be difficult to derive such formulas in this case. This is due to the fact that the divisor reduction involves two loop iteration of Cantor reduction. The resulting expression are really complicated with a lot of multiplications and additions.

Consequently we focus on Fan and Gong projective coordinates [9]. We derive a projective formula for Add32 from the affine formula as follows: we substitute the variable $u_0, u_1, u_2, v_0, v_1v_2$ and u'_0, u'_1, v'_0, v'_1 with $(U_0/Z, U_1/Z, U_2/Z, V_0/Z, V_1/$ and $(U_0/Z, U_1/ZV_0/Z, V_1/Z)$, respectively. Afterwards we arrange the resulting expression in order to minimize the number of field operations. This process is mostly technical so we skip it and provide directly the resulting formula in Algorithm 12 of the appendix.

D. Complexity and implementation results

Complexity comparison. In this subsection we first evaluate the complexity of the proposed approach and compare it to the Double-and-add-always approach. In Table IV-D we provide the complexity of the addition and doubling formulas on $Jac(H(\mathbb{F}_p))$ in affine and projective coordinates. We can notice that the proposed formula for Add32 has 26 multiplications/squarings (resp. 27) less than ADD33 formula in affine coordinates (resp. projective coordinates).

The complexity of the Double-and-add-always in affine coordinates is equal to ℓ times the cost of Add33Aff and Doubling33Aff which gives $\ell(2I + 135M)$. The complexity of the Double-and-add-always in projective coordinates is equal to ℓ times the cost of ADD33Proj and Double33Proj plus a final conversion from projective to affine coordinates which results in a total of $\ell(196M + 20S) + I + 6M$ operations.

For the proposed approach the cost is equal to the cost of a splitting (cf. Table V) plus the cost of ℓ loop iterations each consisting in one Doubing33Aff and one Add32Aff (resp. one Doubling33Proj and one Add32Proj) in affine (resp. projective) coordinates. In projective coordinates a final conversion to affine coordinates is required. The resulting complexity in affine coordinates is as follows

$$\ell(103M + D + 5S + 165a + 2I) + (\frac{9\log_2(p)}{2} + 135)M + (9\log_2(p) + 42)S + 256a + 7I$$

and in projective coordinates we have the following:

$$(179M + 2D + 13S + 156a)\ell + (\frac{9\log_2(p)}{2} + 141)M + (9\log_2(p) + 42)S + 256a + 8I.$$

In Table IV-D we report these complexities under the assumption that the key length $\ell = 3\log_2(p)$. We notice that the improvement in this case is a bit better than for genus 2: we save $20M/S$ per loop iteration when using affine coordinates and around $50M/S$ per loop iteration in projective coordinates.

Operation	Coordinate Syst.	Cost
Add33Aff	Affine	$64M + 3S + 109a + 1I$
Add32Aff	Affine	$39M + D + 2S + 56a + I$
Add31Aff	Affine	$17M + 4S + 33a + I$
Doubling3Aff	Affine	$61M + 7S + 96a + 1I$
MixedAdd33Proj	Projectif	$116M + D + 6S + 102a$
MixedAdd32Proj	Projectif	$63M + D + 7S + 54a$
Doubling3Proj	Projectif	$120M + 6D + 13S + 114a$
Double-and-add-always	Affine	$\ell(125M + 10S + 205a + 2I)$
Proposed Algorithm 5	Affine	$\ell(104.5M + D + 8S + 165a + 2I) + 135M + 42S + 256a + 7I$
Double-and-add-always	Projectif	$\ell(236M + 7D + 19S) + 216a) + I + 6M$
Proposed Algorithm 5	Projectif	$\ell(180.5M + 2D + 16S + 156a) + 141M + 42S + 256a + 8I$

Implementation results. We implemented in C the Double-and-add-always and the proposed approaches for the security level of $192, 244$ and 300 in affine and projective coordinates. We chose three hyperelliptic curves defined over the field $\mathbb{F}_p$ with $p = 2^{73} - 19$ and $p = 2^{94} - 19$ and $p = 2^{113} - 5$, respectively. The implementations strategies are essentially the same as the one used for genus 2 hyperelliptic curves. Indeed, the form of the primes p are essentially the same, so we implemented arithmetic modulo p using the same method. We adapted the divisor splitting to the specificity to the splitting of divisor over genus 3 curves, but the underlying operations (multiplications, additions and exponentiations) are essentially the same. At the end we obtain the timings reported in Table VI. We can notice that most of the time the proposed approach with divisor splitting is better that its Double-and-add-always counterpart. Indeed in affine coordinates we have an improvement which is in the range 5%-15% in affine coordinates, most of the time it is around 7-8%. Generally, projective coordinates is the best choice, and in this case we have an im provements in the range of 17-18%.

Table VI

TIMINGS OF SCALAR MULTIPLICATION OVER GENUS 3 HYPERELLIPTIC CURVES

Method	Coord.	Security level	Timing (10^3 clocks-cycles)		
			Intel Core i5-3210M	Intel Xeon E5-2650v4	ARM
Double-and-add-always	Aff.	192	4233	1440	5366
Proposed	Aff.	192	3868	1298	4756
Double-and-add-always	Proj.	192	3574	1091	5691
Proposed	Proj.	192	2935	906	4630
Double-and-add-always	Aff.	244	5610	1957	8443
Proposed	Aff.	244	5319	1856	7813
Double-and-add-always	Proj.	244	4000	1244	8494
Proposed	Proj.	244	3369	1081	7055
Double-and-add-always	Aff.	300	7602	2661	14458
Proposed	Aff.	300	7080	2468	12219
Double-and-add-always	Proj.	300	5069	1638	17540
Proposed	Proj.	300	4181	1331	14371

We can also notice that if we compare the timings in Table VI with the ones of Table VI it appears that, for the same level of security, a scalar multiplication over a genus 3 curve is generally not competitive compared to a scalar multiplication over a genus 2 curve. This is mostly due to the complexity of divisor operation on genus 3 curves. Until now not much optimization where done on genus 3 curves on doubling and addition of divisor, while more works were done on genus 2 curves.

V. CONCLUSION

In this paper we considered scalar multiplication over genus 2 and genus 3 hyperelliptic curves immune to simple power analysis and timing attacks. We extended the approach of [24] to scalar multiplication over hyperelliptic curves. This approach requires to split the base divisor into a difference of two divisors of smaller degree: two divisors of degree 1 on genus 2 and two divisors of degree 2 on genus 3. We also provided improved formula to add a divisor of degree 2 (resp. degree 3) with a divisor of degree 1 (resp. degree 2) on the Jacobian of a genus 2 (resp. genus3) hyperelliptic curves. This approach reduces the cost of the loop iteration of the scalar multiplication compared to the Double-and-add-always, but this requires a divisor splitting. The complexity evaluation and timing results showed that the proposed approach is interesting in most cases. The reduction in the timings are around 5% to 8% on genus 2 curves and around 15% over genus 3 curves.

REFERENCES

[1] The GNU Multiple Precision Arithmetic Library (GMP). http://gmplib.org/.

[2] M. Brown, D. Hankerson, J. L., and A. Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In *Topics in Cryptology - CT-RSA 2001,*, volume 2020 of *LNCS*, pages 250–265. Springer, 2001.

[3] D. G. Cantor. Computing in the Jacobian of a hyperelliptic curve. *Mathematics of computation*, 48(177):95–101, 1987.

[4] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal Correlation Analysis on Exponentiation. In *ICICS 2010*, volume 6476 of *LNCS*, pages 46–61. Springer, 2010.

[5] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman and Hall/CRC, 2005.

[6] J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES*, pages 292–302, 1999.

[7] C. Costello and K.E. Lauter. Group Law Computations on Jacobians of Hyperelliptic Curves. In *SAC 2011*, volume 7118 of *LNCS*, pages 92–117. Springer, 2012.

[8] S. Duquesne. Montgomery Scalar Multiplication for Genus 2 Curves. In *ANTS-VI*, volume 3076 of *LNCS*. Springer, 2004.

[9] X. Fan, T.J. Wollinger, and G. Gong. Efficient explicit formulae for genus 3 hyperelliptic curve cryptosystems. Technical Report 38, CACR, University of Waterloo, 2006.

[10] P. Gaudry. Fast genus 2 arithmetic based on Theta functions. *Journal of Mathematical Cryptology*, 1(3):243–265, 2007.

[11] R. Harley. Fast arithmetic on genus 2 curves. `http://cristal.inria.fr/harley/hyper` for C source code and further explanations.

[12] H. Hisil and C. Costello. Jacobian Coordinates on Genus 2 Curves. In *ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 338–357. Springer, 2014.

[13] M. Joye and M. Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 334–349. Springer, 2009.

[14] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[15] N. Koblitz. Hyperelliptic Cryptosystems. *J. Cryptology*, 1(3):139–150, 1989.

[16] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[17] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology, CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[18] T. Lange. *Efficient arithmetic on hyperelliptic curves*. PhD thesis, Universität-Gesamthochschule Essen, 2001.

[19] T. Lange. Formulae for Arithmetic on Genus 2 Hyperelliptic Curves. *Appl. Algebra Eng. Commun. Comput.*, 15(5):295–328, 2005.

[20] A. Langley. C25519 code. http://code.google.com/p/curve25519-donna/, 2008.

[21] N. Meloni. New Point Addition Formulae for ECC Applications. In *WAIFI 2007*, volume 4547 of *LNCS*, pages 189–201. Springer, 2007.

[22] V. Miller. Uses of elliptic curves in cryptography. In *Crypto'85*, LNCS. Springer-Verlag, 1985.

[23] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, january 1987.

[24] C. Negre and T. Plantard. Efficient regular modular exponentiation using multiplicative half-size splitting. *J. Cryptographic Engineering*, 7(3):245–253, 2017.

[25] J. Nyukai, K. Matsuo, J. Chao, and S. Tsujii. On the resultat computation in the addition Harley algorithms on hyperelliptic curves. Technical Report ISEC2006-5, IEICE, 2006.

[26] D. Shanks. Five Number-theoretic Algorithms. In *Second Manitoba Conference on Numerical Mathematics*, pages 51–70, 1973.

[27] C. D. Walter. Sliding windows succumbs to big mac attack. In *CHES 2001*, volume 2162 of *LNCS*, pages 286–299. Springer, 2001.

[28] S. Weintraub. The Jacobi Symbol and a Method of Eisenstein For Calculating It, 2000.

[29] Y.-H. Wu, R. Zuccherat, and A. Menezes. *An Elementary Introduction to Hyperelliptic Curves*, pages 155–178. Springer, 1998.

Appendix

Validity of proposed expression for Add21Proj

We prove that the formula in (3) leads to $(U_1'', U_0'', , V_1'', V_1'', Z'', W'')$ such that

$$(U_1''/Z''^2, U_0''/Z''^4, V_1''/(Z''^3W''), V_1''/(Z''^5W'')) = (u_1'', u_0'', u_1'', u_0'')$$

where $(u_1'', u_0'', u_1'', u_0'')$ are the coefficients computed by Algorithm 8.

- *Expression of* l_2, l_1 *in terms of* A, B *and* L_1. First, we can check that, up to a factor of ZW, the fraction A/B is equal to l_2 of (1)

$$\begin{aligned} A/B &= (V_0' - V_0 + U_0' \cdot V_1) \,/\, ((U_0' - U_1) \cdot U_0' + U_0) \\ &= \left(\tfrac{V_0' - V_0 + U_0' \cdot V_1}{Z^5 W}\right) / \left(\tfrac{(U_0' - U_1) \cdot U_0' + U_0}{Z^5 W}\right) \\ &= \left(\tfrac{V_0'}{Z^5 W} - \tfrac{V_0}{Z^5 W} + \tfrac{U_0'}{Z^2} \cdot \tfrac{V_1}{Z^3 W}\right) / \left(((\tfrac{U_0'}{Z^2} - \tfrac{U_1}{Z^2}) \cdot \tfrac{U_0'}{Z^2} + \tfrac{U_0}{Z^4})\tfrac{1}{ZW}\right) \\ &= ((u_0' v_1 + v_0' - v_0)ZW) \,/\, \left(u_0'^2 + u_0 - u_1 u_0'\right) = l_2 ZW \end{aligned}$$

  Then we show that L_1 is equal to l_1 up to the factor $(BZW)^3$

$$\begin{aligned} L_1/(BZW)^3 &= (BV_1(BW)^2 + AU_1(BW)^2)/(BZW)^3 \\ &= V_1/(WZ^2) + (U_1/Z^2)(A/BZW) = v_1 + u_1 l_2 = l_1; \end{aligned}$$

- *Expression* u_1'' *and* u_0'' *in terms of* A, B, L_1 *and* U_1''. We set $Z'' = BZW$ and $U_1'' = A^2 - (U_1 + U_0')BW$. then we can check that U_1''/Z''^2 is equal to the term u_1'' in (2):

$$\begin{aligned} U_1''/Z''^2 &= U_1''/(BZW)^2 = A^2/(BZW)^2 - (U_1/Z^2 + U_0'/Z^2), \\ &= -l_2^2 - (u_1 + u_0'). \end{aligned}$$

  Now we consider

$$U_0'' = -U_0(BW)^4 - ((U_1(BW)^2) \times U_0'(BW)^2) + (BW)^4 \times f_3 Z^4 - U_1''(U_1(BW)^2 + U_0'(BW)^2) - 2L_1(BW)^2 A.$$

  We can check that U_0''/Z''^4 is equal to the affine expression u_0'' in (2):

$$\begin{aligned} U_0''/(BZW)^4 &= -U_0/Z^4 - (U_1/Z^2) \times (U_0'/Z^2) + f_3 - (U_1''/(BZW)^2)(U_1/Z^2 + U_0'/Z^2) \\ &\qquad -2(L1/(BZW)^3)(A/(BZW)) \\ &= -u_0 - (u_1 u_0') + f_3 - u_1''(u_1 + u_0') - 2l_1 l_2 \end{aligned}$$

- *Expression of* v_1'' *and* v_0'' *in terms of* V_0'' *and* V_1''. We consider V_0'' and V_1'' of (3). We can check that V_1''/Z''^3 and V_0''/Z''^5 are equal to the terms v_1'' and v_0'' computed in Algorithm 8:

$$\begin{aligned} V_1''/(Z'')^3 &= (AU_1'' - L1)/(BZW)^3 = l_2 u_1'' - l_1 = v_1'' \\ V_0''/(Z'')^5 &= (A(U_0'' - U_0(BW)^2) - V_0 B^4 W^5)/(BWZ)^5 = l_2(u_0'' - u_0) - v_0 = v_0'' \end{aligned}$$

  And this ends the proof.

Conversion formula : ZW Jacobian coordinates $\leftrightarrow$ affine coordinates.

- Degree 2 case (Convert2_AfftoZW). Given $D = (X^2 + u_1 X + u_0, v_1 X + v_1)$ and Z, W we get the expression of $D = (U_1, U_0, V_1, V_0, Z, W)$ as follows

$$\begin{aligned} U_1 &= u_1 \times Z^2 && \text{with } 1S + 1M \\ U_0 &= u_0 \times (Z^2)^2 && \text{with } 1S + 1M \\ V_1 &= v_1 \times (Z^2 \times (Z \times W)) && \text{with } 3M \\ V_0 &= v_0 \times Z^2 \times (Z^3 W) && \text{with } 2M \end{aligned}$$

and the total cost is $7M + 2S$

- Degree 1 case (Convert1_AfftoZW). Given $D = (X + u_0, v_0)$ and Z, W we have to compute

$$\begin{aligned} U_0 &= u_0 \times (Z^2) && \text{with } 1S + 1M \\ V_0 &= v_0 \times (Z^2)^2 \times Z \times W && \text{with } 1S + 3M \end{aligned}$$

which requires $2S + 4M$ to get $D = (U_0, V_0, Z, W)$.

Formulas for genus 3 hyperelliptic curves

We provide in the remaining of the appendix the formulas over the Jacobian of a genus 3 hyperelliptic curve. We first present the formula Add32Aff for the addition of a degree 3 divisor with a degree 2 divisor in affine coordinates which is based on the discussion in Subsection IV-C. We also present the formula AddAff31 for the addition of a degree 3 divisor with a degree 1 divisor: this formula is used in the splitting of the divisor (Algorithm 10) and was obtained with the method of Costello and Lauter [7]. We present the formulas for Add33Proj and Doubling3Proj: these formulas are corrected versions of the formulas provided in [9] which contain several errors. For completeness sake we recall the Add33Aff and Doubling3Aff which are from [9].

Algorithm 11 Add32Aff(u0,u1,u2,v0,v1,v2,z,uu0,uu1,vv0,vv1)

```
//-------  coeff matrix a,b,c,d ------------------------- Cost = 7M+14a
tp0:=(uu1-u2)*uu0;   a:=u0+tp0;   tp1:=(u2-uu1)*(u2+uu1);    tp2:=(tp1-u1);
tp3:=u0*u2;   b:=-tp3+(tp2+uu0)*uu0;   c:=-uu0+u1+(uu1-u2)*uu1;  tp4:=u1*u2;
d:=-tp4+u0+(tp2+2*uu0)*uu1;
//-------   Delta, inverse of Delta and  l4   ----------- Cost = 10M+6a+I
Delta := a*d-c*b; coeff1:=(vv0 - v0 + uu0 * v2); coeff2:=(vv1 - v1 + uu1*v2);
ll4:=( -c * coeff1 + a*coeff2);   tp5:=ll4*Delta;   Itp5:=1/tp5;
IDelta:=ll4*Itp5;   Ill4:=Delta*Itp5;  Il4:=Delta*Ill4;
//---- l3ol4=l3/l4, l1ol4=l1/l4 and l2ol4=l2/l4 --------- Cost = 8M+7a
l3ol4:= (  d * coeff1 - b * coeff2)*Ill4; l4:=IDelta*ll4;
l1ol4:= u1 * l3ol4 + (u0-tp4) + v1*Il4; l2ol4:=u2*(l3ol4-u2)+u1+v2*Il4;
//-----UUU --------------------------------------------- Cost = 7M+D+2S+20a
tp6:=  (uu0 + u1 + uu1*u2);  tp7:=uu0*u2+uu1*u1; tp8:=(Il4)^2;
uuu2 := 2*(l3ol4)-tp8 -uu1-u2;  uuu1:=2*l2ol4+(l3ol4)^2 - tp6 - uuu2*(uu1+u2);
uuu0:=2*l1ol4 + 2*(l3ol4)*(l2ol4)-f5*tp8 -((tp7+u0)+tp6*uuu2+(uu1+u2)*uuu1);
//---- VVV --------------------------------------------- Cost = 7M+9a
vvv0 := ((uuu0-u0) * l3ol4 + tp3  - uuu0*uuu2)*l4-v0;
vvv1 := (-l1ol4+uuu0+uuu1*(-uuu2+ l3ol4))*l4;
vvv2 := (-l2ol4+uuu1+uuu2*(-uuu2 + l3ol4))*l4;
return uuu0,uuu1,uuu2,vvv0,vvv1,vvv0;     //--- Overall cost = 39M+2S+D+I+56a
```

Algorithm 12 Add32Proj(u0,u1,u2,v0,v1,v2,z,uu0,uu1,vv0,vv1)

```
//-------  coefficients  a,b,c,d  and determinant ------ Cost = 14M+2S +13a
tp0:=uu1*z;  tp1:=(tp0-u2)*uu0; a:=u0+tp1; tp1:=u2^2; tp2:=uu0*z; tp3:=u0*u2;
tp4:=tp0^2;b:=-tp3+(-u1+tp2)*tp2+uu0*(tp1-tp4); c :=(uu1*(tp0-u2)-tp2 +u1);
tp5:=u1*z; tp6:=u0*z; tp7:=u1*u2; d:=(2*tp0*tp2-tp7+tp6+(tp1-tp4-tp5)*uu1);
Delta := (a*d-c*b);
//-------  L3 and L4 ----------------------------------- Cost = 9M+6a
coeff1:=(vv0*z-v0+uu0*v2); coeff2:=(vv1*z - v1 +uu1*v2);
L3 := (d*coeff1-b*coeff2); LL4 := (-c*coeff1+a*coeff2);  L4 :=  LL4*z ;
//-------- L0,L1 and L2 -------------------------------- Cost = 10M+8a
zD:=z*Delta;  L0 := (tp6 * L3 -tp3 * L4 + v0*zD) ;
L1:=(tp5*L3+(tp6-tp7)*L4+v1*zD);  L2 := (u2*z * L3 +(tp5-tp1) * L4 +v2*zD);
//-------- UUU ----------------------------------------- Cost = 20M+D+4S+20a
tp8:=LL4^2;  tp9:=L4*L3;  tp10:=Delta^2; tp11:=tp8*z;
Tuuu2 := (((2*tp9-tp10)+(-tp0-u2)*tp11)); uuu2:=Tuuu2*z;
tp12:=LL4*L2;  tp13:=L3^2*z;  tp14:=uu1*u2;  tp15:= tp11*z ;
Tuuu1 :=2*tp12+tp13 - (tp2+u1+tp14)*tp15 -Tuuu2*(tp0+u2);  uuu1:=Tuuu1*z;
z2:=z^2;   tp19:=tp11*z2;tp16:=L4*L1;  tp17:=L3*L2;  tp18:=tp10*z2*f5;
Tuuu0:=((2*(tp16+tp17)-tp18);
Tuuu0:=Tuuu0-(((uu0*u2+uu1*u1+u0)*tp19+(tp2+u1+tp14)*uuu2)+(tp0+u2)*Tuuu1));
//------ VVV and ZZZ  ---------------------------------- Cost =10M+5a
tp20:=(tp9 - Tuuu2); Tvvv0 := (-L0 *tp15*L4 +Tuuu0 * tp20);
Tvvv1 := ((-tp16+Tuuu0)*tp15 + uuu1*tp20);
Tvvv2 := ((-tp12+Tuuu1)*tp11 + Tuuu2*tp20);
L4D:=L4*Delta;  tp21:=z2*L4D; ZZZ:=(tp15*tp21);
return Tuuu0,uuu1,Tuuu2,Tvvv0,Tvvv1,Tvvv2,ZZZ;//----Total cost =63+6S+1D+54a
```

Algorithm 13 Add31Aff(u10,u11,u12,v10,v11,v12,u20,v20)

```
//------------------------------------------------- Cost = 5M+S+6a+I
w0:=u20^2;    w1:=w0*(u20-u12);   w2:=u11*u20;
r:=-(w1+w2-u10);    i:=r^(-1);  z0:=u20*v12;
s0:=i*(v20-u20*(z0-v11)-v10);
//------ u32 --------------------------------------- Cost = S+2a
t0:=s0^2;    t1:=u20+u12; u32:=t0+t1;
//------ u31 --------------------------------------- Cost = 4M+7a
t2:=u12*u20; t3:=t2+u11; t4:=s0*u12; t5:=2*s0*(t4+v12)-f5;u31:=(t5+t3-t1*u32);
//------ u30 --------------------------------------- Cost = 5M+2S+10a
t6:=w2+u10; t7:=s0*u11; t8:=u12*v12;
t9:=t4^2+2*s0*(t7+t8+v11)+v12^2-f4; u30:=t9+t6-t3*u32-t1*u31;
//------ v32,v31,v30 -------------------------------- Cost = 3M+8a
v32:=-(s0*(u32+u12)+v12); v31:=-(s0*(u31+u11)+v11);  v30:=-(s0*(u30+u10)+v10);
return u30,31,u32,v30,v31,v32   -------------------- Cost = 17M+4S+33a+I
```

Algorithm 14 MixedAdd33Proj(U10,U11,U12,V10,V11,V12,Z1,U20,U21,U22,V20,V21,V22) [9]

```
//------------ Precomputations  -------------------------- Cost = 6M
UU22:=Z1*U22; UU21:=Z1*U21; UU20:=Z1*U20; VV22:=Z1*V22; VV21:=Z1*V21;
VV20:=Z1*V20;
//------------- Resultant of U1 and U2 ------------------- Cost = 15M+13a
t0:=U10-UU20; t1:=U11-UU21;  t2:=U12-UU22;  t3:=t1-U22*t2;
t4:=t0-U21*t2; t5:=t4-U22*t3; t6:=U20*t2+U21*t3;  t7 :=-(t4*t5+t3*t6);
t8:=t2*t6+t1*t5; t9:=t2*t4-t1*t3; r :=t0*t7-U20*(t3*t9+t2*t8);
//------------- Pseudo inverse S'=rS=(V2-V1)I mod U2------ Cost = 10M+26a
i2:=t9; i1 := t8;  i0 := t7; t1 :=V10-VV20; t2 :=V11-VV21; t3:=V12-VV22;
t4:=t2*i1;  t5:=t1*i0; t6:=t3*i2; t7:=U22*t6; t8:=t4+t6+t7 - (t2+t3)*(i1+i2);
t9:=U20+U22 ; t10:=(t9 + U21)*(t8 - t6); t9:=(t9 - U21)*(t8 + t6);
ss0:=-(U20*t8+t5); ss1:=(t4+t5-t7) - (t1+t2)*(i0+i1) + (t9-t10)/2;
ss2:=(t6-t4-ss0) - (t1+t3)*(i0+i2) - (t9+t10)/2;
//------------- Precomputations--------------------------- Cost = 9M+4S
w0 := ss0*Z1; w1:=ss1*Z1; w2:=ss2*Z1; w3:=ss2^2; w4:=w2^2; R:=r*Z1; RR:=r^2;
A:=w3*Z1; B:=R*w2;  D:=B*R;  E:=B^2;  F:=w2*Z1; G:=E*F;
//------------- Computation of Z-------------------------- Cost = 11M+15a
z0:=ss0*U10 ; z1:=(ss0+ss1)*(U10+U11)-ss1*U11-ss0*U10;
z2:=(ss0+ss2)*(U10+U12)-ss2*U12-ss0*U10+ss1*U11;
z3:=w0+(ss1+ss2)*(U12+U11)-ss1*U11-ss2*U12; z4:=w1+ss2*U12;
//------------- Computation of Ut------------------------- Cost = 25M+27a
ut3:=z4+w1-U22*w2; t1:=ss1*z4-(ss2*ut3)*U22; ut2:=ss2*(z3+w0-U21*w2)+t1;
t2:=(U22+U21)*(ss2*ut3+ut2); t3:=(ss0*z3-U21*ut2); t4:=z2+r*V12; t5:=z1+r*V11;
ut1:=ss2*(t4+r*V12)+(ss0+ss1)*(z3+z4)-r*R-(t1+t2+t3+A*U20);
ut0 :=ss2*(t5+r*V11)+ss1*(t4+r*V12)+t3+RR*U12-(ss2*ut3)*U20-U22*ut1;
//------------- Computation of Vt------------------------- Cost = 12M+8a
t1 :=ut3-z4; vt0:=t1*ut0+A*(z0+V10*r); vt1:=t1*ut1+w2*(ss2*t5-ut0);
vt2 := t1*ut2+w2*(ss2*t4-ut1); vt3:=t1*ut3+w2*z3-ut2*Z1;
//------------- Computation of Z3 and U3------------------ Cost = 22M+D+2S+10a
t1:=2*vt3; u32:=-(D*ut3+vt3^2); u31:=D*( w4*f5-ut2*Z1)-(u32*ut3+t1*(vt2*Z1));
u30:=E*(w4*f4-ut1*Z1)-((vt2*Z1)^2+u32*(ut2*Z1)+u31*ut3+t1*F*vt1);
u32:=u32*w4; u31:=u31*w2 ; Z3:=G*w4*r; U32:=u32*B; U31:=u31*B; U30:=u30*B;
//------------- Computation of V3------------------------- Cost = 6M+3a
V32:=G*vt2-u32*vt3; V31:=G*vt1-u31*vt3; V30:=G*vt0-u30*vt3;
return(U30,U31,U32,V30,V31,V32,Z3) //------------- Total cost = 116M+6S+D+102a
```

Algorithm 15 Doub33Proj(U10,U11,U12,V10,V11,V12,Z1) [9]

```
//----- Precomputations ------------------------------- Cost = 6M+S
Z:=Z1^2; UU12:=Z1*U12; UU11:=Z1*U11; UU10:=Z1*U10; VV12:=Z1*V12; VV11:=Z1*V11;
VV10:=Z1*V10;
//--- Resultant  r and pseudo inverse i2,i1,i0--------- Cost = 17M+10a
t1:=VV11-U12*V12; t2:=VV10-U11*V12; t3:=Z1*t2-U12*t1; t4:=UU10*V12+U11*t1;
t5:=t2*t3+t1*t4; t6:=-(V11*t3+V12*t4); t7:=V11*t1-V12*t2;
r:=Z*(VV10*t5-UU10*(t1*t7+V12*t6)); i2:=t7;i1:=t6;i0:=t5;
//----- Computation of Z ------------------------------ Cost = 7M+4D+3S+19a
t1:=U12^2;t2:=f4*Z-(2*UU10+V12^2);t3:=f5*Z+t1-2*UU11; z2:=Z*(t3+2*t1);
z1:=UU12*(2*UU11-t3)+Z*t2;
z0:=f3*Z^2+t1*(t3-UU11)+UU12*(2*UU10-t2)+UU11*(UU11-f5*Z)-2*VV12*VV11;
//----- Computation of SS and  A,B,C,D,E,F------------- Cost = 19M+2S+28a
t1:=i1*z1; t2:=i0*z0; t3:=i2*z2; t4:=U12*t3;
t5:=(i1+Z1*i2)*(z1+z2)-(t1+Z1*t3+t4);  t6:=U10*t5; t7:=U10+U12; t8:=t7+U11;
t9:=t7-U11; t7:=t8*(Z1*t3+t5); t11:=t9*(t5-Z1*t3);
ss2:=Z1*(t1-Z1*t3)+t6+(i0+Z*i2)*(z0+z2)-(t2+(t7+t11)/2);
ss1:=Z1*(t4-t1)+(i0+Z1*i1)*(z0+z1)+(t11-t7)/2-t2; ss0:=t2-t6;
A:=ss2^2;B:=2*r*ss2;D:=2*r*B;E:=B^2;F:=A*D;
//----- Computation of G ------------------------------ Cost = 9M+15a
g0:=ss0*U10; g1:=(ss0+ss1)*(U10+U11)-ss1*U11-ss0*U10;
g2:=(ss0+ss2)*(U10+U12)-ss2*U12-ss0*U10+ss1*U11;
g3:=ss0*Z1+(ss1+ss2)*(U12+U11)-ss1*U11-ss2*U12; g4:=ss1*Z1+ss2*U12;
//----- Computation of Ut and Vt ---------------------- Cost = 29M+2S+27a
ut3:=2*ss1; ut2:=ss1^2+2*ss0*ss2; ut1:=2*(ss1*ss0*Z1+2*r*(ss2*V12-r*Z1));
ut0:=ss0^2*Z+4*r*(U12*(2*r*Z1-ss2*V12)+ss1*VV12+ss2*VV11); t1:=ut3*Z1-g4;
vt0:=t1*ut0+Z*A*(g0+2*r*V10);vt1:=t1*ut1+Z1*A*(g1+2*r*V11)-ss2*ut0;
vt2:=t1*ut2+A*(g2+2*r*V12)-ss2*ut1; vt3:=t1*ut3-ut2*Z1+ss2*g3;
//----- Computation of Z2 and U2 ---------------------- Cost = 24M+2D+4S+12a
t0:=Z1^2; t1:=2*vt3; u22:=-(D*ut3*t0+vt3^2);
u21:=D*(f5*A-ut2)*t0-(u22*ut3+t1*vt2);
u20:=E*(f4*A*Z1-ut1)*t0-((vt2^2+u22*ut2+u21*ut3)*Z1+ss2*t1*vt1);
u21:=u21*ss2; u22:=u22*A; t2:=2*r*ss2^2; t3:=Z1*t2; t4:=Z1*t3;t5:=t3^2;
Z2:=t5*t4; U22:=u22*t4;  U21:=u21*t4;  U20:=u20*t3; t6:=ss2*vt3;
//----- Computation of V2 ----------------------------- Cost = 9M+1S+3a
V22:=(t5*vt2-u22*t6)*Z1;  V21:=t5*vt1-u21*Z1*t6; V20:=Z1*t2^2*vt0-u20*t6;
return(U30,U31,U32,V30,V31,V32,Z3) //---------------- Cost = 120M+6D+13S+114a
```

Algorithm 16 Add33Aff(u10,u11,u12,v10,v11,v12,u20,u21,u22,v20,v21,v22)

```
// Etape 1 --- resultant  -------------------------------- Cost = 15M+12a
t0:=u10-u20; t1:=u11-u21; t2:=u12-u22; t3:=t1-u22*t2;  t4:=t0-u21*t2;
t5:=t4-u22*t3;  t6:=u20*t2+u21*t3; t7:=-(t4*t5+t3*t6); t8:=t2*t6+t1*t5;
t9:=t2*t4-t1*t3; r:=t0*t7-u20*(t3*t9+t2*t8);
// Etape 3 --- pseudo inverse I
i2:=t9; i1:=t8; i0:=t7;
// Etape 4 --- S'---------------------------------------- Cost = 10M+30a
t1:=v10-v20; t2:=v11-v21; t3:=v12-v22; t4:=t2*i1; t5:=t1*i0; t6:=t3*i2;
t7:=u22*t6 ; t8:=t4+t6+t7-(t2+t3)*(i1+i2); t9:=u20+u22; t10:=(t9+u21)*(t8-t6);
t9:=(t9-u21)*(t8+t6);
ss0:=-(u20*t8+t5); ss1:=t4+t5+(t9-t10)/2-(t7+(t1+t2)*(i0+i1));
ss2:=t6-(ss0+t4+(t1+t3)*(i0+i2)+(t9+t10)/2);
// Etape 6 ---------------------------------------------- Cost = 6M+S+1I
t1:=(r*ss2)^(-1); t2:=r*t1; w:=t1*ss2^2; wi:=r*t2;s0:=t2*ss0; s1:=t2*ss1;
// Etape 7 ---  Z --------------------------------------- Cost = 4M+15a
t1:=u10+u12; t2:=(s0+s1)*(t1+u11); t3:=(s0-s1)*(t1-u11); t4:=u12*s1;
z0:=u10*s0; z1:=(t2-t3)/2-t4; z2:=(t2+t3)/2-z0+u10;
z3:=u11+s0+t4;z4:=u12+s1;
// Etape 8 ---  Ut -------------------------------------- Cost = 13M+27a
ut3:=z4+s1-u22; t1:=s1*z4-u22*ut3; ut2:=z3+s0+t1-u21; t2:=(u22+u21)*(ut3+ut2);
t3:=s0*z3-u21*ut2;  ut1:=z2+(s0+s1)*(z4+z3)+wi*(2*v12-wi)-(t1+t2+t3+u20);
ut0:=z1+t3+s1*z2+wi*(2*(v11+s1*v12)+wi*u12)-(u22*ut1+u20*ut3);
// Etape 9  ---  Vt ------------------------------------- Cost = 8M+11a
t1:=ut3-z4; vt0:=w*(t1*ut0+z0)+v10; vt1:=w*(t1*ut1+z1-ut0)+v11;
vt2:=w*(t1*ut2+z2-ut1)+v12; vt3:=w*(t1*ut3+z3-ut2);
// Etape 10 ---  U3 ------------------------------------- Cost = 5M+2S+11a
t1:=2*vt3; u32:=-(ut3+vt3^2); u31:=f5-(ut2+u32*ut3+t1*vt2);
u30:=f4-(ut1+vt2^2+u32*ut2+u31*ut3+t1*vt1);
// Etape 11 - calcul de V3-------------------------------- Cost = 3M+3a
v32:=vt2-u32*vt3; v31:=vt1-u31*vt3; v30:=vt0-u30*vt3;
return(u32,u31,u30,v32,v31,v30);//---------------  Total cost =I+ 64M+3S+109a
```

Algorithm 17 Doubling33Aff(U10,U11,U12,V10,V11,V12,Z1)

```
//---------- Step 1, Resultant ---------------------------- Cost = 15M+10a
t1 :=v11-u12*v12; t2:=v10-u11*v12; t3:=t2-u12*t1; t4:=u10*v12+u11*t1;
t5:=t2*t3+t1*t4; t6:=-(v11*t3+v12*t4); t7:=v11*t1-v12*t2;
r:=v10*t5-u10*(t1*t7+v12*t6);
//---------- Step 3 -- Pseudo inverse  --------------------
i2:=t7; i1:=t6; i0:=t5;
//---------- Step 4 -- S':=z02X^2+z1X+z0:=(F-V1^2)/U1------ Cost = 5M+2S+16a
t1:=2*u10; t2:=2*u11; t3:=u12^2; t4:=f4-(t1+v12^2);
t5:=f5+t3-t2; t10:=2*v12; z2:=t5+2*t3; z1:=u12*(t2-t5)+t4;
z0:=f3+t3*(t5-u11)+u12*(t1-t4)+u11*(u11-f5)-t10*v11;
//---------- Step 5 -- S'=s'2X^2+s'1X+s'0=2rS=ZI mod U1---- Cost = 10M+22a
t1:=i1*z1; t2:=i0*z0; t3:=i2*z2; t4:=u12*t3; t5:=(i1+i2)*(z1+z2)-(t1+t3+t4);
t6:=u10*t5; t7:=u10+u12; t8:=t7+u11; t9:=t7-u11; t7:=t8*(t3+t5);
t11:=t9*(t5-t3);  ss0:=t2-t6; ss1:=t4+(i0+i1)*(z0+z1)+(t11-t7)/2-(t1+t2);
ss2:=t1+t6+(i0+i2)*(z0+z2)-(t2+t3+(t7+t11)/2);
//---------- Step 7 ------- S := S'/2r -------------------- Cost = 6M+S+1a+I
t1:=2*r; t2:=(t1*ss2)^(-1); t3:=t1*t2; w:=t2*ss2^2;
wi:=t1*t3; s0:=t3*ss0; s1:=t3*ss1;
 //--------- Step 8 ------- computation of G = SU1 -------- Cost = 4M+12a
t1:=t8*(s0+s1); t2:=t9*(s0-s1); t3:=u12*s1;
g0:=u10*s0; g1:=(t1-t2)/2-t3; g2:=(t1+t2)/2-g0+u10; g3:=u11+s0+t3; g4:=u12+s1;
//---------- Step 9----- Ut = ((G+wiV1)^2-wi^2F)/U1^2------ Cost = 5M+2S+10a
ut3:=2*s1; ut2:=s1^2+2*s0; ut1:=ut3*s0+wi*(t10-wi);
ut0:=s0^2+2*wi*((s1-u12)*v12+v11+wi*u12);
//---------- Step 10 --- computation of (wG+V1) mod Ut----- Cost = 8M+11a
t1:=ut3-g4; vt0:=w*(t1*ut0+g0)+v10; vt1:=w*(t1*ut1+g1-ut0)+v11;
vt2:=w*(t1*ut2+g2-ut1)+v12; vt3:=w*(t1*ut3+g3-ut2);
//---------- Step 11 --- Computation of U2 ---------------- Cost = 5M+2S+11a
t1:=2*vt3; u22:=-(ut3+vt3^2); u21:=f5-(ut2+u22*ut3+t1*vt2);
u20:=f4-(ut1+vt2^2+u22*ut2+u21*ut3+t1*vt1);
//---------- Step 12 --- calcul de V2  -------------------- Cost = 3M+a
v22:=vt2-u22*vt3; v21:=vt1-u21*vt3; v20:=vt0-u20*vt3;
return(u22,u21,u20,v22,v21,v20);//----------- Total Cost = I+61M+7S+96a
```