Faster Constant-time Evaluation of the Kronecker Symbol with Application to Elliptic Curve Hashing

Diego F. Aranha
dfaranha@cs.au.dk
Aarhus University
Aarhus, Denmark

Benjamin Salling Hvass
bsh@cs.au.dk
Aarhus University
Aarhus, Denmark

Bas Spitters
spitters@cs.au.dk
Aarhus University
Aarhus, Denmark

Mehdi Tibouchi
mehdi.tibouchi.br@hco.ntt.co.jp
NTT Corporation
Tokyo, Japan

ABSTRACT

We generalize the Bernstein-Yang (BY) algorithm [BY19] for constant-time modular inversion to compute the Kronecker symbol, of which the Jacobi and Legendre symbols are special cases. We first develop a basic and easy-to-implement algorithm, defined with full-precision division steps. We then describe an optimized version due to Hamburg [Ham21] over word-sized inputs, and formally verify its correctness. Along the way, we introduce a number of optimizations for implementing both versions in constant time. The resulting algorithms are particularly suitable for computing the Legendre symbol with dense prime p, where no efficient addition chain is known for exponentiating to $\frac{p-1}{2}$, as it is often the case in pairing-friendly elliptic curves. Our high-speed implementation for a range of parameters shows that the new algorithm is up to 40 times faster than exponentiation, and up to 25.7% faster than the previous state of the art. We illustrate our techniques with hashing to elliptic curves using the SwiftEC algorithm [CRT22], with savings of 14.7% – 48.1%, and to accelerating the CTIDH isogeny-based key exchange [BBC+21], with savings of 3.5 – 13.5%.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**.

KEYWORDS

Kronecker/Jacobi/Legendre Symbol • Division step • Constant-time software implementation • Formal verification.

1 INTRODUCTION

Many algorithms of cryptographic interest in Number Theory can be expressed as variants of the Euclidean algorithm. Natural examples employed in many cryptosystems are algorithms for computing modular inversion and testing if integers have a square root with respect to a modulus (quadratic residuosity). Respectively, these algorithms are used to generate keys in factorization-based cryptosystems, such as RSA and Paillier, or to convert elliptic curve points from projective to affine coordinates; and for testing validity of the x-coordinate of an elliptic curve point when hashing.

In its original form, the Euclidean algorithm executes a variable number of iterations computed over monotonically decreasing inputs until a certain condition is met. This aspect of the algorithm, however, poses a challenge for its implementation in cryptographic contexts, when both security and performance matter. This is true under a threat model where the adversary is capable of measuring characteristics of the running implementation, such as execution time or power consumption, mounting a so-called *side-channel attack*. In such a setting, an adversary can observe irregular patterns in the implementation and collect leakage that reveals sensitive information, for example bits of private inputs [ASS17, AGTB19, AGB20]. A particular class of side-channel attacks that is considered easier to exploit consists of *timing attacks*, in which an adversary collects leakage by measuring timing differences locally or remotely. Common countermeasures against such attacks are employing *constant-time* implementation or employing additional randomness to make input values or execution time less predictable.

The literature has many hardened variations of Euclidean algorithms that behave in a more regular way and reduce the amount of leakage. The simplest approach is to evaluate the functions computed by these algorithms using one or more exponentiations by a fixed power depending on the modulus, for example $(p-2)$ for inversion modulo a prime p. When parameters permit, this can be evaluated efficiently in constant-time using a short addition chain. Alternatively, a branchless implementation could evaluate all targets of branches and conditionally select only the correct one at each iteration [Bos14]. This would effectively eliminate timing attacks based on branch prediction, but at high performance penalty. Another common trick is to introduce a blinding factor that randomizes the execution, which might keep the algorithm in variable-time but decorrelates the leakage from the actual inputs [ASS17]. In many cases, these protections impose a non-trivial performance penalty that greatly reduce the efficiency of implementations in comparison to the unprotected versions; or improve security at most heuristically by modestly increasing attack complexity, while not addressing the root causes for side-channel leakage.

Related work. The Bernstein-Yang (BY) algorithm for modular inversion [BY19] is a recent development that elegantly improved solutions available to this problem. It consists of a fast algorithm that can be implemented in constant time, since it is formulated in terms of *division steps* that can be efficiently evaluated in a regular manner. The algorithm is quite flexible, and can be generalized for polynomial arithmetic or other problems related to the Euclidean algorithm. Recent work by Hvass et al. [HAS23] has formally verified the theory underlying the BY algorithm using a foundational

approach, and synthesized efficient and formally verified implementations for inclusion in the Fiat-Crypto framework [EPG+19]. An improvement called the *half-delta* optimization [WMO21] reduces the number of iterations by approximately 18% on average.

In a recent independent preprint [Ham21], Hamburg continues this line of work and proposes a variation of the algorithm to compute the Jacobi symbol $\left(\frac{a}{b}\right)$ in constant time, for integers a, b with odd positive b. The Jacobi symbol is a generalization of the Legendre symbol and, from its definition, can be computed by factoring $b = \prod_i p_i^{e_i}$ and multiplying the Legendre symbols $\left(\frac{a}{p_i}\right)^{e_i}$ for each prime $p_i > 2$ and multiplicity e_i. In turn, the Legendre symbol represents the quadratic residuosity of a modulo p_i and can be computed as $\left(\frac{a}{p_i}\right) \equiv a^{\frac{p_i-1}{2}} \pmod{p_i}$. There are Euclidean algorithms to evaluate both symbols faster than computing exponentiations, but they are hard to implement efficiently in constant-time.

From another angle, Pornin recently introduced a new algorithm to compute modular inversion in constant time by unrolling the inner loop in a variation of the binary Euclidean algorithm [Por20b]. He later adapted it for the Legendre symbol as well, and presented timings for an implementation targeting ARM Cortex-M0 microcontrollers [Por20a]. This work represents the current state of the art, and will server as the reference point for comparison.

Contributions. In this paper, we generalize the full-precision version of the BY algorithm to compute the Kronecker symbol, which generalizes both the aforementioned Legendre and Jacobi symbols. We note that the Legendre and Jacobi symbols arise more frequently in practice, and that the Legendre symbol with respect to a fixed second argument (prime modulus) is the case of higher interest to cryptography. This case arises when testing the quadratic residuosity of a private numerator against a public prime modulus.

When restricted to the Legendre case, we obtain a significant speedup over the exponentiation approach for dense prime moduli, where a short addition chain to exponentiate by $\frac{p-1}{2}$ is not known and modular multiplication is more expensive due to Montgomery arithmetic. The case of dense prime moduli frequently occur in pairing-based cryptography, where the finite fields are defined in terms of prime numbers parameterized by polynomials, that quickly grow in density when evaluated over sparse integers. Moreover, we optimize Hamburg's faster word-sized version of the algorithm [Ham21] and obtain a further speedup over the exponentiation approach. The correctness of our algorithms is backed by a formal verification of the underlying theory using the framework developed in [HAS23], so it also implies correctness of [Ham21].

Our experimental treatment focuses on applications to cryptography, so we evaluate the Legendre symbol in constant time with respect to multiple choices of prime modulus. Our efficient implementation and experiments point out that the basic and faster versions of the algorithm are up to 7 and 40 times faster than the exponentiation approach when implemented in constant time for several prime moduli underlying popular elliptic curves, and up to 25.7% over the previous state of the art established in [Por20a]. We also explore an application of the algorithm for testing quadratic residuosity within the SwiftEC approach to hash arbitrary strings to points on an elliptic curve, obtaining an approximate speedup ranging from 14.7% to 48.1% depending on the curve.

Another application is to constant-time variants of the quantum-safe CSIDH isogeny-based key agreement scheme [CLM+18], such as CTIDH [BBC+21] and SQALE [CCJR22]. Current implementations use the exponentiation approach with a dense prime modulus, so they should benefit. When our techniques are applied to CTIDH, observed savings amount to 3.5–13.5%.

Organization. The paper is organized as follows. Section 2 gives the mathematical background about the various symbols and the original BY algorithm. Section 3 develops the new algorithms and optimizations. Section 4 presents our benchmarking approach and experimental results, and Section 5 concludes.

2 PRELIMINARIES

In this section, we review definitions for the Legendre, Jacobi and Kronecker symbols and their main properties. These are functions applied to integer parameters taken from increasingly larger sets. In textbooks, all three symbols use the same notation $\left(\frac{a}{b}\right)$, but we will denote them with L, J and K subscripts to make explicit what symbol we are referring to. We note that this does not change their definitions in any way, but it will prove important later on when we introduce the algorithmic parts and move across different parameter ranges.

2.1 Basic definitions

The Legendre symbol of a over p, written as $\left(\frac{a}{p}\right)_L$ for integer a and odd prime p, is defined as:

$$\left(\frac{a}{p}\right)_L = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p}, \\ 1 & \text{if } a \not\equiv 0 \pmod{p} \text{ and } \exists x \in \mathbb{Z} : a \equiv x^2 \pmod{p}, \\ -1 & \text{otherwise.} \end{cases}$$

In other words, the Legendre symbol encodes the information about a being a quadratic residue modulo p, and thus can be evaluated through Euler's criterion by computing $\left(\frac{a}{p}\right)_L \equiv a^{\frac{p-1}{2}} \pmod{p}$.

The Legendre symbol satisfies several properties:

- *Periodicity in the first argument (numerator):* if $a \equiv b \pmod{p}$ then $\left(\frac{a}{p}\right)_L = \left(\frac{b}{p}\right)_L$
- *Complete multiplicativity:* $\left(\frac{ab}{p}\right)_L = \left(\frac{a}{p}\right)_L \left(\frac{b}{p}\right)_L$.

In particular, it allows to state the *law of quadratic reciprocity* for odd primes p and q:

$$\left(\frac{p}{q}\right)_L \left(\frac{q}{p}\right)_L = (-1)^{\frac{p-1}{2}\frac{q-1}{2}},$$

and its two supplements:

$$\left(\frac{-1}{p}\right)_L = (-1)^{\frac{p-1}{2}} = \begin{cases} 1 & \text{if } p \equiv 1 \pmod{4}, \\ -1 & \text{if } p \equiv 3 \pmod{4}. \end{cases}$$

$$\left(\frac{2}{p}\right)_L = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 & \text{if } p \equiv \pm 1 \pmod{8}, \\ -1 & \text{if } p \equiv \pm 3 \pmod{8}. \end{cases}$$

The Jacobi symbol generalizes the Legendre symbol to odd positive integers b, given the factorization of $b = \prod_i p_i^{e_i}$ for $p_i > 2$:

$$\left(\frac{a}{b}\right)_J = \prod_i \left(\frac{a}{p_i}\right)_L^{e_i}.$$

It also satisfies many of the same properties of the Legendre symbol, for example periodicity of the first argument, and complete multiplicativity in one argument when the other is fixed. Like the Legendre symbol, if $\left(\frac{a}{b}\right)_J = -1$ then a is a quadratic nonresidue modulo b. The contrapositive $\left(\frac{a}{b}\right)_J = 1$ for quadratic residue a modulo b is only true if a and b are coprime. Hence it also satisfies a more general law of quadratic reciprocity with supplements, expressed in the same way but for odd positive coprime integers.

The Kronecker symbol further generalizes the Jacobi symbol to all remaining cases, where $b = u \cdot \prod_i p_i^{e_i}$ for $u \in \{-1, 0, 1\}$ and $p_i \geq 2$, defined as:

$$\left(\frac{a}{0}\right)_K = \begin{cases} 1 & \text{if } a = \pm 1, \\ -1 & \text{otherwise.} \end{cases} \qquad \left(\frac{a}{-1}\right)_K = \begin{cases} 1 & \text{if } a \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

$$\left(\frac{a}{2}\right)_K = \begin{cases} 0 & \text{if } a \text{ is even,} \\ 1 & \text{if } a \equiv \pm 1 \pmod 8, \\ -1 & \text{otherwise.} \end{cases} \qquad \left(\frac{a}{b}\right)_K = \left(\frac{a}{u}\right)_K \prod_i \left(\frac{a}{p_i}\right)_K^{e_i}$$

The Kronecker symbol shares many of the basic properties of the Jacobi symbol, but under more restrictions, and does not have the same connection to quadratic residuosity as the previous two. In this work, we will consider the most general definition of the Kronecker symbol, but note that some textbooks restrict the definition to $b > 0$ for simplicity [Ros95].

2.2 The Bernstein-Yang algorithm

The BY algorithm for modular inversion relies on the definition of a *division step* that updates the operands as the algorithm executes for a fixed number of iterations [BY19]. A division step (divstep) is defined for all integers δ, g and odd integers f as:

$$\text{divstep}(\delta, f, g) = \begin{cases} \left(1 - \delta, g, \frac{g-f}{2}\right) & \text{if } \delta > 0 \text{ and } g \text{ odd,} \\ \left(1 + \delta, f, \frac{g+(g \bmod 2)f}{2}\right) & \text{otherwise.} \end{cases}$$

The algorithm also computes *transition matrices*:

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ odd,} \\ \begin{pmatrix} 2 & 0 \\ g \bmod 2 & 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Note that we use the definition of $\mathcal{T}$ from [HAS23] which differs slightly from the presentation in [BY19]. For integers δ, f and g, we write $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.

Algorithm 1 iterates the divstep function, computing and sequentially multiplying the transition matrix of the resulting values. For a constant-time implementation, the branch can be implemented by just looking at individual bits of δ and g, the sign flips by converting the corresponding variables in two's-complement representation, and the assignments by conditional swaps to exchange the values when the branch is taken (lines 4-5 in the algorithm).

Algorithm 2 implements modular inversion using Divsteps by computing the number of iterations required (lines 1-5), setting up the constants (lines 6-7), iterating the division steps the required number of times (line 8) and combining the results at the end (line 9). In the algorithm, $\text{sgn}(\cdot)$ computes the sign of an integer. For a fixed modulus f and assuming $1 < g < f$, the case commonly occurring for modular arithmetic in cryptography, we can precompute the values n and e in advance. The correctness of the algorithm is given by the main theorem in [BY19], reproduced below.

Theorem 2.1 (Theorem 11.2 in [BY19]). *Let f and g be integers with f odd. Let d be a real number such that $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$. Let m be an integer such that $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$ and $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$.*

For $i = 1, 2, \ldots, m$, let $(\delta_i, f_i, g_i) = \text{divstep}^i(1, f, g)$ and $\mathcal{T}_i = \mathcal{T}(\delta_i, f_i, g_i)$ and $\begin{pmatrix} u_i & v_i \\ q_i & r_i \end{pmatrix} = \mathcal{T}_{i-1}\mathcal{T}_{i-2} \cdots \mathcal{T}_0$. Then $g_m = 0$, $f_m = \pm \gcd(f, g)$ and $v_m g = 2^m f_m \pmod f$.

Since f and g are assumed to be coprime, the final values of f and v are respectively f_m and v_m, and p is the inverse of 2^m modulo f, so the following holds:

$$p \cdot v \cdot \text{sgn}(f) \cdot g = (2^{-m})v(\pm 1)g = (\pm 1)(\pm 1) = 1 \pmod f.$$

Again we use the presentation from [HAS23] which differs slightly from the one in [BY19] because of our definition of $\mathcal{T}$.

Algorithm 1: Divsteps for inversion

Input : Integers n, δ, f and g such that f is odd
Output : The integers δ_n, f_n and g_n and the matrix product $\mathcal{T}_n\mathcal{T}_{n-1} \cdots \mathcal{T}_0$

1 $u \leftarrow 1, v \leftarrow 0, q \leftarrow 0, r \leftarrow 1$
2 **for** $i \leftarrow 1$ **to** n **do**
3 **if** $0 < \delta$ and g odd **then**
4 $\delta \leftarrow -\delta, f \leftarrow g, g \leftarrow -f,$
5 $u \leftarrow q, v \leftarrow r, q \leftarrow -u, r \leftarrow -v$
6 $g_0 \leftarrow g \bmod 2, \delta \leftarrow \delta + 1$
7 $q \leftarrow q + g_0 u, r \leftarrow r + g_0 v$
8 $g \leftarrow \frac{g+g_0 f}{2}, u \leftarrow 2u, v \leftarrow 2v$
9 **return** $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$

3 ALGORITHMS FOR KRONECKER SYMBOL

We describe two versions of the algorithm: an easier-to-implement full-precision version based on divstep, described in Section 3.1; and a faster word-sized version described in Section 3.2. Like with modular inversion, we will assume that the first argument to Divsteps is public. It corresponds to the prime modulus in inversion, and the bottom argument in the Kronecker symbol. Hence the algorithms can leak the values of specific bits of f, and the lengths of both arguments f and g, since the number of iterations depend directly on those. Since f is known in advance and $0 \leq g < f$ in the cases of interest within cryptography, there is no impact on security.

Algorithm 2: Bernstein-Yang modular inversion algorithm.

Input : Integers f and g such that f is odd and $\gcd(f,g) = 1$
Output: Integer g^{-1} such that $gg^{-1} = 1 \pmod f$

$d \leftarrow \max(\text{bits}(f), \text{bits}(g))$
if $d < 46$ **then**
  $n \leftarrow \lfloor (49d + 80)/17 \rfloor$
else
  $n \leftarrow \lfloor (49d + 57)/17 \rfloor$
$e \leftarrow ((f+1)/2)^n \bmod f$
$\delta \leftarrow 1$
$\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right) \leftarrow \textsc{Divsteps}(n, \delta, f, g)$
return $e \cdot v \cdot \text{sgn}(f)$

3.1 Full-precision Divsteps version

We start by defining a symbol that extends the Jacobi symbol to include negative numbers in the denominator as $\left(\frac{a}{b}\right) := \left(\frac{a}{|b|}\right)_J$, for integers a, b. Even numbers are also handled by factoring out powers of $\left(\frac{a}{2}\right)_K = \left(\frac{2}{a}\right)_K$ under multiplicativity. This is well-defined, so we have the following version of reciprocity for coprime a and b:

$$\left(\frac{a}{b}\right)\left(\frac{b}{a}\right) = \varepsilon(a,b)(-1)^{\frac{a-1}{2}\frac{b-1}{2}}, \tag{1}$$

where $\varepsilon(a,b) = -1$ if both a and b are negative and 1 otherwise. All other properties of the usual Jacobi symbol are preserved in this extended symbol, such as multiplicativity in both arguments and periodicity in the numerator.

Now we extend the divstep function to also record information about the value of the Jacobi symbol. For $k \in \{\pm 1\}$ we define:

$$\text{divstep}_k(\delta, f, g, k) = \begin{cases} \left(1-\delta, g, \frac{g-f}{2}, (-1)^{\frac{g^2-1}{8}}\varepsilon(g,-f)(-1)^{\frac{g-1}{2}\frac{-f-1}{2}}k\right) & \text{if } \delta > 0 \text{ and } g \text{ odd}, \\ \left(1+\delta, f, \frac{g+(g \bmod 2)f}{2}, (-1)^{\frac{f^2-1}{8}}k\right) & \text{otherwise.} \end{cases}$$

In this first case, we swap (f,g) with $(g,-f)$ and employ the laws of quadratic reciprocity to handle even f and multiplicatively accumulate the result into the intermediate value k. In the second case, we apply the second supplement of the law of quadratic reciprocity to handle even g and accumulate the result. Now we can prove that the recurrence computes the extended symbol correctly.

Lemma 3.1. *For all $n \in \mathbb{N}$ write $(\delta_n, f_n, g_n, k_n) = \text{divstep}_k^n(\delta, f, g, k)$. If we take $k = 1$ and assume that $\gcd(f,g) = 1$, then we have*

$$k_n\left(\frac{g_n}{f_n}\right) = \left(\frac{g}{f}\right).$$

In particular, if $f_n = \pm 1$, then $k_n = \left(\frac{g}{f}\right)$.

Proof. We proceed by induction in n. By assumption $k_0 = k = 1$, so the formula is true when $n = 0$.

If $\delta_{n+1} > 0$ and g_{n+1} is odd, then

$$\begin{aligned} k_{n+1}\left(\frac{2g_{n+1}}{f_{n+1}}\right) &= (-1)^{\frac{g_n^2-1}{8}}\varepsilon(g_n,-f_n)(-1)^{\frac{g_n-1}{2}\frac{-f_n-1}{2}}k_n\left(\frac{g_n-f_n}{g_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)k_n\left(\frac{g_n}{-f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)\left(\frac{g}{f}\right), \end{aligned}$$

and the result follows by dividing by $\left(\frac{2}{f_{n+1}}\right)$. Note that we use that f_{n+1} is odd and that f_n and g_n are coprime (since f and g are assumed to be so).

If $\delta_{n+1} \le 0$ or g_{n+1} is even, then

$$\begin{aligned} k_{n+1}\left(\frac{2g_{n+1}}{f_{n+1}}\right) &= (-1)^{\frac{f_n^2-1}{8}}k_n\left(\frac{g_n + (g_n \bmod 2)f_n}{f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)k_n\left(\frac{g_n}{f_n}\right) \\ &= \left(\frac{2}{f_{n+1}}\right)\left(\frac{g}{f}\right). \end{aligned}$$

and the result follows as before. □

By the main theorem of [BY19], there is an m such that $(f_m, g_m) = (\pm\gcd(f,g), 0)$ so we get the following corollary.

Lemma 3.2. *Let f and g be integers with f odd. We have that*

$$\left(\frac{g}{f}\right) = \begin{cases} k_m & \text{if } f_m = \pm 1 \\ 0 & \text{otherwise.} \end{cases}$$

This means that if we iterate divstep_k as many times as is needed for divstep to converge, then we also get an algorithm for computing the extended symbol $\left(\frac{g}{f}\right)$ defined previously.

For simplicity, we will split the resulting algorithm for arbitrary integers f, g into two different subroutines, hereby referred to as inner and outer. The inner routine will evaluate the extended symbol $\left(\frac{g}{|f|}\right)$ by iterating divstep_k the required number of times. Algorithm 3 does exactly that, but it also computes the full transition matrix as a side-effect. Algorithm 4 is a modification to evaluate the branches in constant time explicitly, and to avoid computing the transition matrix that is not needed for the actual evaluation of the symbol. We define an operator $\text{msb}(\cdot)$ to evaluate the most-significant bit in two's complement representation that coincides exactly with the sign evaluation.

The other general cases corresponding to the Kronecker symbol will be handled in the outer Algorithm 5. Assuming that f is a public parameter, the algorithm can be trivially modified to be executed in constant-time with respect to the value of g, the parameter assumed to be private. We handle the case $f = 0$ right after counting the number of iterations, and use a temporary variable u to correct the sign when both f and g are negative and when f is even. We argue that Algorithm 5 does not leak information about g when the branches are evaluated in constant time, other than its length when iterating $\textsc{Divsteps}_k$ for the corresponding number of iterations. In the cases of interest to cryptography, g is assumed to be reduced modulo f, so the number of iterations is determined by f alone and

does not leak information about g. We can further modify Algorithm 5 to run in constant-time when both f and g are private and non-trivial at some performance loss by adding dummy iterations of the loop (line 14), given an upper bound on the power of 2 dividing f.

Algorithm 3: Divsteps$_k$ for Kronecker symbol

Input : Integers n, δ, f and g such that f is odd
Output: The integers δ_n, f_n, g_n and k_n

$k \leftarrow 0, u \leftarrow 1, v \leftarrow 0, q \leftarrow 0, r \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  if $\delta > 0$ and g odd **then**
    $\delta \leftarrow -\delta, f \leftarrow g, g \leftarrow -f$
    $k \leftarrow k + ((\lfloor \frac{f}{2} \rfloor \bmod 2) \cdot (\lfloor \frac{g}{2} \rfloor \bmod 2) + 1) \bmod 2$
    if $f < 0$ and $g < 0$ **then**
      $k \leftarrow k + 1 \bmod 2$
  $g_0 \leftarrow g \bmod 2, \delta \leftarrow \delta + 1$
  $k \leftarrow k + ((\lfloor \frac{f}{2} \rfloor \bmod 2) \cdot (\lfloor \frac{f}{4} \rfloor \bmod 2)) \bmod 2$
  $q \leftarrow q + g_0 u, r \leftarrow r + g_0 v$
  $g \leftarrow \frac{g + g_0 f}{2}\ u \leftarrow 2u, v \leftarrow 2v$
return $\delta, f, g, k, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$

Algorithm 4: Optimized OptDivsteps$_k$ in constant time

Input : Integers n, δ, f and g such that f is odd
Output: The integers δ_n, f_n, g_n and k_n
Note : All conditional copies and swap ($\leftrightarrow$) must be implemented in constant time with bit operations.

$k \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $d_0 \leftarrow (\delta > 0) \wedge (g \bmod 2)$
  $\delta \leftarrow -\delta$ **if** $(d_0 > 0)$
  $f \leftrightarrow g$ **if** $(d_0 > 0)$
  $g \leftarrow -g$ **if** $(d_0 > 0)$
  $k \leftarrow k \oplus (((f \wedge g) \gg 1) \oplus (\mathrm{msb}(f) \wedge \mathrm{msb}(g)) \wedge d_0)$
  $g_0 \leftarrow g \bmod 2, \delta \leftarrow \delta + 1$
  $k \leftarrow k \oplus ((f \gg 1) \bmod 2) \wedge ((f \gg 2) \bmod 2)$
  $g \leftarrow (g + g_0 f) \gg 1$
$t \leftarrow 2k$ `// adjust output for Jacobi symbol`
return $\delta, f, g, 1 - t$

3.2 Word-oriented JumpDivsteps version

Algorithm 1 can be optimized by observing that computing the ℓ first iterations of Divsteps only depends on the ℓ first bits in f and g. This allows working on smaller numbers and "jumping" through the iterations of Divsteps in larger steps (see Section 10 in [BY19]).

This optimization, however, can unfortunately not be naively applied to Algorithm 4. The issue is that going to a lower bit length by truncating, one loses information about the most significant bits of f and g. Fortunately, a trick discovered by Hamburg in [Ham21] can be used to circumvent this issue: in order to compute the contribution of $(\mathrm{msb}(f) \wedge \mathrm{msb}(g))$ to k output by Algorithm 4, one only needs to compute the amount of times f changes sign. Now this in itself is still not enough, since f is truncated. Instead it turns out that counting the amount of times the value q changes sign in Algorithm 3 is sufficient for computing the contribution of sign changes from f.

Algorithm 5: Kronecker symbol based on Divsteps

Input : Integers f and g
Output: Kronecker symbol $\left(\frac{g}{f}\right)_K$

$d \leftarrow \max(\mathrm{bits}(f), \mathrm{bits}(g))$
if $d < 46$ **then**
  $n \leftarrow \lfloor (49d + 80)/17 \rfloor$
else
  $n \leftarrow \lfloor (49d + 57)/17 \rfloor$
if $f = 0$ **then**
  if $g = \pm 1$ **then**
    return 1
  else
    return 0
$u \leftarrow 1$
if $(f < 0)$ ***and*** $(g < 0)$ **then**
  $u \leftarrow -1$ `// handle negative operands`
while $f \bmod 2 = 0$ **do**
  $f = f/2$
  if $g \bmod 2 = 0$ **then**
    $u \leftarrow 0$ `// return 0 no matter k below`
  if $g \bmod 8 = \pm 3$ **then**
    $u \leftarrow -u$ `// factor out (g|2) and flip u`
$\delta \leftarrow 1$
$\delta', f', g', k \leftarrow$ OptDivsteps$_k(n, \delta, |f|, g)$
if $|f'| \neq 1$ **then**
  return 0
else
  return $u \cdot k$

Hamburg defines a 2 by 2 integer matrix to be *ratchet*, if it has strictly positive determinant and its second row is positive, and then proceeds to prove two theorems.

Theorem 3.3 (Theorem 1, [Ham21]). *Let M_i be a sequence of ratchet matrices and let $T_i = \left(\begin{smallmatrix} a_i & b_i \\ c_i & d_i \end{smallmatrix}\right)$ be the sequence of matrices recursively defined by $T_0 = I$ and $T_i = M_i \cdot T_{i-1}$. Let $f, g \in \mathbb{Z}$ with either $f \neq 0$ or $g > 0$ and write $\left(\begin{smallmatrix} f_i \\ g_i \end{smallmatrix}\right) = T_i \cdot \left(\begin{smallmatrix} f \\ g \end{smallmatrix}\right)$. If t_j, u_j denote the number of times that f_i, c_i change signs for $i \leq j$, respectively, then $t_j - u_j \in \{0, 1\}$.*

Theorem 3.4. *For all $n \in \mathbb{N}$, $\mathcal{T}_n$ is ratchet.*

Using these two theorems, one can apply the jumpdivstep optimization to Algorithm 3 to define a variant of Algorithm 4 which does not use the most significant bits of f or g, but rather counts the sign changes of q to compute the same contribution. Let us refer to this algorithm as JumpDivsteps$_k$, corresponding to the extended

Jacobi symbol computation introduced in [Ham21]. Our version, optimized for constant-time execution, in depicted in Algorithm 6. For a processor with word size w, an optimal value for ℓ would be the largest integer smaller than $(w-2)$ that divides the required number of iterations, but approximations are also possible.

Algorithm 6: JumpDivsteps$_k$ for Kronecker symbol

Input : Integers n, ℓ, w, δ, f and g such that f is odd and positive, and $\ell \leq (w-2)$
Output: The integers δ_n, f_n, g_n and k_n
Note : All branches, conditional copies and selects must be implemented in constant time.

$k \leftarrow 0$
for $i \leftarrow 1$ **to** $\lceil n/\ell \rceil$ **do**
  $f' \leftarrow f \bmod 2^w, g' \leftarrow g \bmod 2^w$
  $u \leftarrow 0$
  $\left(\begin{smallmatrix} a_i & b_i \\ c_i & d_i \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$
  for $j \leftarrow 1$ **to** ℓ **do**
    $y \leftarrow f', d_0 \leftarrow (\delta \geq 0), m_1 \leftarrow (g' \bmod 2)$
    $m_0 \leftarrow d_0 \wedge m_1$
    $t_0 \leftarrow f', t_1 \leftarrow c_i, t_2 \leftarrow d_i$
    if $(d_0 > 0)$ **then**
      $t_0 \leftarrow -t_0, t_1 \leftarrow -t_1, t_2 \leftarrow -t_2$
    $g' \leftarrow g' + t_0$ **if** $(m_1 > 0)$
    $a_i \leftarrow a_i + t_1$ **if** $(m_1 > 0)$
    $b_i \leftarrow b_i + t_2$ **if** $(m_1 > 0)$
    $f' \leftarrow f' + g'$ **if** $(m_0 > 0)$
    $c_i \leftarrow c_i + a_i$ **if** $(m_0 > 0)$
    $d_i \leftarrow d_i + b_i$ **if** $(m_0 > 0)$
    $g' \leftarrow g' \gg 1, c_i \leftarrow 2c_i, d_i \leftarrow 2d_i$
    $\delta \leftarrow 2 - \delta$ **if** $(m_0 > 0)$ **or** $2 + \delta$ **if** $(m_0 = 0)$
    $u \leftarrow u + ((y \wedge f') \oplus (f' \gg 1)) \wedge 2$
    $u \leftarrow u + ((u \wedge 1) \oplus \mathrm{msb}(c_i))$
  $\left(\begin{smallmatrix} g \\ f \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} a_i & b_i \\ c_i & d_i \end{smallmatrix}\right) \cdot \left(\begin{smallmatrix} g \\ f \end{smallmatrix}\right) / 2^\ell$
  $k \leftarrow (k+u) \bmod 4$
  $k \leftarrow k + ((k \bmod 2) \oplus \mathrm{msb}(f)) \bmod 4$
$t \leftarrow (k + (k \bmod 2)) \bmod 4$
return $\delta, f, g, 1-t$

Now we are in position to define Algorithm 7 as an adaptation of Algorithm 5 that uses the modified JumpDivsteps$_k$. It computes the number of iterations using the half-delta optimization [WMO21], that reduces the number of iterations by approximately 18% on average, then handles the corner cases just like Algorithm 5. It finishes by invoking JumpDivsteps$_k$ as a subroutine.

Formal Verification of Hamburg's Paper. At the time of writing, Hamburg's result [Ham21] has not been formally peer-reviewed yet. Thus, we formalize his work as a public attestation of its correctness. To do so, we build on top of the certified proof of the correctness of the BY inversion algorithm in the Coq Proof Assistant [HAS23]. To be precise, we formalize the results which are necessary for the correctness of the JumpDivsteps implementation of the Kronecker computation.

Algorithm 7: Kronecker symbol based on JumpDivsteps

Input : Integers f and g, word size w
Output: Kronecker symbol $\left(\frac{g}{f}\right)_K$

$d \leftarrow \max(\mathrm{bits}(f), \mathrm{bits}(g))$
$n = \lfloor (45907d + 26313)/19929 \rfloor$
if $f = 0$ **then**
  if $g = \pm 1$ **then**
    return 1
  else
    return 0
$u \leftarrow 1$
if $(f < 0)$ ***and*** $(g < 0)$ **then**
  $u \leftarrow -1$ `// handle negative operands`
while $f \bmod 2 = 0$ **do**
  $f = f/2$
  if $g \bmod 2 = 0$ **then**
    $u \leftarrow 0$ `// return 0 no matter k below`
  if $g \bmod 8 = \pm 3$ **then**
    $u \leftarrow -u$ `// factor out (g|2) and flip u`
$\delta \leftarrow 1, \ell \leftarrow w - 2$
$\delta', f', g', k \leftarrow$ JumpDivsteps$_k(n, \ell, w, \delta, |f|, g)$
if $|f'| \neq 1$ **then**
  return 0
else
  return $u \cdot k$

To formalize Theorem 3.3 we need some auxiliary functions. We define $\mu : \mathbf{Z} \times \mathbf{Z} \to \{0, 1\}$ by $\mu(a, b) = 1$ if a and b have different signs and 0 otherwise; this is used to count sign changes. We also define ε as in section 3.1. Using these we can rephrase Theorem 3.3 as follows.

Theorem 3.5. *Let $i \in \mathbb{N}$ and c_j and f_j be defined as in Theorem 3.3 for all $j \in \{0, \ldots, i\}$. Then*

$$\sum_{j=0}^{i-1} \mu(f_{j+1}, f_j) = \sum_{j=0}^{i-1} \mu(c_{j+1}, c_j) + (\mu(f_i, f_0) + \mu(c_i, c_0) \bmod 2).$$

The formalization of this proposition in Coq is the following:

```
Definition ratchet_spec M i :=
  κ M i =
    (κ' M i) + ((μ (f M i) (f M 0)) + (μ (c M i) (c M 0))) mod 2.
```

where κ and κ' correspond to the sums in Theorem 3.5.

The proof proceeds much like in [Ham21]. We split each matrix M_i into a product of three matrices, each of which are either upper triangular or rotation matrices.

We implement the decomposition as a map t from sequences of matrices to sequences of matrices, defined in Coq as

```
Definition t (M : nat -> mat Q) :=
  fun (i : nat) =>
    let '(a, b, c, d) := M (i / 3)%nat in
    if (decide (b ≡ 0))
    then match mod3dec i with
         | inright _ => I
         | inleft pf => match pf with
                        | right _ => I
                        | left _ => (a, b, c, d)
                        end
         end
    else match mod3dec i with
         | inright _ => (b, 0, d, 1)
         | inleft pf => match pf with
                        | right _ => (0, 1, -1, 0)
                        | left _ => (a * d / b - c, 0, a / b, 1)
                        end
         end.
```

where `mod3dec` decides whether a number is 0, 1 or 2 modulo 3. Note that all the matrices in the sequence $t(M)$ are either upper triangular or rotation matrices. In Coq we have that:

```
Lemma t_lt_or_rot M : forall i, lt_or_rot (t M i).
```

We prove that the sequence $t(M)$ actually decomposes the sequence M, such that multiplying three consecutive matrices in $t(M)$ (starting at a multiple of 3) gives a corresponding matrix in M:

$$(t(M))_{3\cdot i+2} \cdot (t(M))_{3\cdot i+1} \cdot (t(M))_{3\cdot i} = M_i.$$

We then prove that Theorem 3.5 is satisfied *if* M is a sequence of matrices which all are upper triangular or rotation matrices. This is the theorem below in Coq:

```
Theorem aux M :
  f0 ≠ 0 \/ g0 > 0 ->
  (forall i, ratchet (M i)) ->
  (forall i, lt_or_rot (M i)) ->
  (forall i, ratchet_spec M i).
```

By applying the decomposition function t, we get Theorem 3.5:

```
Theorem thm M :
  f0 ≠ 0 \/ g0 > 0 ->
  (forall i, ratchet (M i)) ->
  (forall i, ratchet_spec M i).
```

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

Our collection of experimental results is split in two parts. In the first subsection we study the performance of the Kronecker symbol in isolation, by benchmarking the Legendre symbol across multiple parameters from curve-based cryptography. In the second, we observe the performance of the algorithm in the context of elliptic curve hashing.

4.1 Legendre symbol

For benchmarking, we specialize to the Legendre symbol, which is the most interesting case of cryptographic interest. We have implemented our algorithms for a range of parameters commonly used in both elliptic curve cryptography (ECC) and pairing-based cryptography (PBC) settings. For the elliptic curve cryptography setting, we chose the 256-bit primes $2^{255} - 19$ and $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ labeled by their respective named curves Curve25519, and `secp256k1` at the 128-bit security level. We note however that the specific prime shape does not affect the performance of our algorithms. For pairing-based cryptography, we selected three different primes to define the base field for Barreto-Lynn-Scott (BLS) curves [BLS02] at multiple security levels. In particular, those are respectively the prime moduli underlying the BLS curves with embedding degree 12 undergoing standardization at 128-bit security [MSS16], the 509-bit prime for BLS curves with embedding degree 24 proposed for 192-bit security [AFK+12, BD19], and the 575-bit prime for BLS curves with embedding degree 48 proposed for 256-bit security [MAF20]. We select a final larger parameter corresponding to the CTIDH 1024-bit prime modulus [BBC+21].

The code was developed in the RELIC toolkit [AGM+] with the C programming language, due to its strong support for curve-based cryptography. For reference, RELIC has a number of pairing-friendly curves implemented efficiently, having set multiple speed records for their computation. There is support for advanced elliptic curve hashing algorithms [WB19] and handwritten Assembly acceleration for the field arithmetic. This approach allowed comparisons between our algorithms for Legendre symbol computation and the other algorithms already implemented in the library.

In the implementation, we followed the library standard and implemented arithmetic using saturated arithmetic, with 64-bit limbs. We also implemented support for Pornin's algorithm [Por20a] by incorporating it from the optimized implementation found in the `blst` [Sup] multi-lingual library implementing Boneh-Lynn-Shacham short signatures [BLS04] over the BLS12-381 elliptic curve. For fairness of comparison, we performed minor tweaks in this implementation to enjoy the same low-level field arithmetic primitives from RELIC as our algorithms.

Benchmarking measurements were taken by computing the average latency of running the code for 10^4 consecutive executions on an Intel Kaby Lake Core i7-7700 CPU at 3.60GHz. The compilers used were GCC version 12.2.1 and `clang` 15.0.7, with optimization flags `-O3 -funroll-loops -march=native -mtune=native`. Following benchmarking conventions [1], TurboBoost and Hyper-Threading were disabled for higher stability.

Our first set of results are presented in Table 1. In the table, the first part sets the baseline for comparison. We included the highly-optimized variable-time implementation of the Jacobi symbol [Möl19] from the GNU MP library version 6.2.1 [Gt20], and the exponentiation approach by Euler's criterion using a generic constant-time algorithm. The particular choice configured for the latter was a sliding-window exponentiation with precomputation table of 32 elements, as per the default RELIC configuration. These two algorithms represent the conventional approaches for variable-time and constant-time implementation, and respectively set both a lower bound for aggressively optimized variable-time code, and an upper bound for generic constant-time approach. The next part brings what is arguably the current state-of-the-art-algorithms for computing the Legendre symbol in constant time. The first line for related work labeled with (C+ASM) contains timings for Pornin's algorithm [Por20a] using RELIC's Assembly acceleration for the various fields. The line labeled with (pure ASM) constains pure Assembly implementations for Pornin's algorithm found in the `blst` library for BLS12-381, or an implementation that we built ourselves.

[1] https://bench.cr.yp.to/supercop.html

In the case of Curve25519 and secp256k1, the pure ASM implementation was built by computing addition chains for $\frac{p-1}{2}$ using [McL21], combined with ASM code for squarings/multiplications from [NS22]. In the case of CTIDH-1024, an addition chain for computing square roots was already found in the original code base. The last part of the table shows the performance of our algorithms, both the basic Divstep approach and the JumpDivstep optimization.

From the baseline entries in the first part of the table, we can observe the massive difference in performance between the two approaches, a factor ranging from 5 to 72. These illustrate how poorly the generic exponentiation-based approach performs across the various parameters. It is at best competitive with the basic Divsteps implementation, but only for the shorter parameters. Starting with the ECC parameters, the sparse primes benefit both the variable-time GMP implementation and the addition-chain based exponentiation approach, exactly as expected. For the constant-time implementations, the related work in pure ASM is the fastest, with a 11% speedup over our JumpDivstep algorithm (8,846 instead of 9,891 cycles, respectively). When we increase the prime length by 1 bit but move to the secp256k1 field, where the modulus is not as sparse and arithmetic is not as efficient, the addition-chain approach is penalized and the numbers are reversed: now JumpDivsteps becomes 11% faster (9,756 instead of 10,971 cycles, respectively). Now moving to the dense pairing-friendly and CTIDH primes, the generic constant-time exponentiation based approach does not scale well and cannot be easily optimized with shorter addition chains, so the basic Divsteps approach becomes competitive by providing a speedup between 1.5 and 7.1 in total.

The real competition is between Pornin's algorithm in (C+ASM) versus JumpDivsteps, in which we obtain a consistent improvement of 17.8-25.7% across all fields. In the particular case of the BLS12-381 prime which is heavily optimized in `blst`, the fully unrolled pure ASM implementation of the same algorithm gives a substantial 42% improvement over the (C+ASM) version, and becomes 23% faster than our JumpDivsteps approach. However, we believe that comparison is not entirely fair and our JumpDivsteps approach would benefit similarly from the same-level of optimization, to the point of providing a similar improvement we obtained in comparison to the (C+ASM) version.

In terms of implementation security, we validated that our compiled code in C+ASM runs in constant-time after compilation by using the `dudect` tool to perform statistical testing of execution time, given random inputs for a fixed prime modulus [RBV17].

4.2 Application to elliptic curve hashing

Hashing arbitrary strings to elliptic curve points is a fundamental operation in many cryptographic protocols, for example the BLS pairing-based short signature scheme [BLS04]. As mentioned in the main standardization effort collecting hash-to-curve constructions, other applications include Password Authenticated Key Exchange (PAKE) protocols, oblivious pseudorandom Functions (OPRFs) and identity-based encryption [FHSS+21].

Various techniques have been proposed to perform hashing efficiently to an elliptic curve in short Weierstrass form $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ over a field of large prime characteristic p, starting from the basic try-and-increment. With this approach, a cryptographic hash function is used to hash the string to the x-coordinate of a point, and then the value $z = x^3 + ax + b$ is tested for quadratic residuosity. In the positive case, the y-coordinate is computed as $\sqrt{z}$ in $\mathbb{F}_p$; otherwise x is incremented and another quadratic residuosity test performed. However, there are several problems with this approach: the distribution of outputs is uncertain; and the algorithm is intrinsically variable-time, which might leak information about the string being hashed. For illustration, timing attacks were recently mounted against variable-time hash-to-curve in the WPA3 protocol [VR20], so efficient constant-time hashing is desirable for real-world applications.

A more principled alternative approach was proposed by Brier et al. [BCI+10]. This approach is composed in two stages, first the message $m \in \{0,1\}^*$ is hashed to a field element in $\mathbb{F}_p$, and then the Shallue-van de Woestijne-Ulas (SWU) encoding $f : \mathbb{F}_p \to E(\mathbb{F}_p)$ is used to map the hash output to a point in the elliptic curve. Given two cryptographic hash functions h_1 and h_2 mapping from arbitrary strings to $\mathbb{F}_p$, the complete process constructs the elliptic curve hash function $H : \{0,1\}^* \to E(\mathbb{F}_p)$ as simply evaluating $H(m) = f(h_1(m)) + f(h_2(m))$, and then multiplying the result by a cofactor to obtain a point in the right prime-order subgroup. From the security point of view, it is known that the map H will satisfy the standard security notion of random oracle indifferentiability when both h_1 and h_2 are indifferentiable from random oracles as well. Unfortunately, this approach is quite expensive in which it needs two evaluations of the encoding map f (plus a minor elliptic curve point addition) to perform its role. Most follow-up work has focused on specializing and optimizing this approach to different classes of elliptic curves, for example BLS12 [WB19], by accelerating the computation of the map f and proposing variants easier to implement in constant time.

The performance drawback of previous approaches was recently improved in the research literature with the SwiftEC [CRT22] algorithm. In this new work, Chávez-Saab et al. reformulate the encoding map f as a parameterized family of functions, such that $f_{h_2(m)}(h_1(m))$ is indifferentiable from a random oracle if h_1 and h_2 are indifferentiable as well. The algorithm constructs a conic S admitting a two-parameterization over $\mathbb{F}_p$, and uses it to obtain three candidate coordinates x_1, x_2, x_3 for the x-coordinate of a point in E, such that at least one of them will be such that $(x_i^3 + ax_i + b)$ is a square. The latter will be exactly the x-coordinate, and a matching y-coordinate is computed by taking a square-root and choosing the sign based on an additional bit. This idea saves the evaluation of one encoding function, which in turn saves on square-root extractions and quadratic residuosity tests. The algorithm is also considerably easier to implement in constant-time than previous work, providing additional benefits beyond just the performance improvement.

In the set of pairing-based cryptography, we are actually interested in hashing to two different groups. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an asymmetric (or Type-3) pairing defined over three groups of prime order r in which the discrete logarithm is hard to compute. It is well known that we can represent $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with subgroups of order r of respectively $E(\mathbb{F}_p), E(\mathbb{F}_{p^k})$ and $\mathbb{F}^*_{p^k}$. Moreover, it is also well-known that we can represent $\mathbb{G}_2$ by an isomorphic subgroup of the same order in the d-degree twist $E'(\mathbb{F}_{p^{k/d}})$ for a more compact and efficient representation. Luckily, it is also simple to instantiate

Table 1: Benchmarks of different approaches for Legendre computation over different fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime. With exception of variable-time GMP, all implementations run in constant time.

	Curve25519		secp256k1		BLS12-381		BLS24-509		BLS48-575		CTIDH-1024	
	gcc	clang	gcc	clang	gcc	clang	gcc	clang	gcc	clang	gcc	clang
Variable-time GMP	6,556	6,553	6,563	6,556	10,375	10,354	13,723	13.827	15,419	15,456	27,918	27,915
Generic constant-time exp.	37,200	36,778	34,264	34,779	87,798	86,502	228,315	226,817	305,544	308,035	1,956,753	2,011,781
Related work (C+ASM)	11,336	11,322	11,271	**11,217**	17,323	17,460	24,357	**24,046**	27,565	**27,419**	**62,633**	65,875
Related work (pure ASM)	8,847	**8,846**	**10,971**	10,979	**10,000**	10,003	–	–	–	–	1,320,345	1,320,358
This work (Divsteps)	35,727	50,321	35,773	49,631	69,164	73,148	88,873	105,857	129,074	129,060	274,551	300,490
This work (JumpDivsteps)	**9,891**	10,277	**9,756**	10,182	**13,044**	14,074	**17,865**	19,101	**22,643**	23,760	**49,182**	50,611

the SwiftEC techniques in curves defined over extension fields, by simply doing the same operations over the larger field instead. We implement the quadratic residuosity test by following the work of Adj and Rodríguez-Henríquez [AR14], which reduces the test in the field $\mathbb{F}_{p^{k/d}}$ to a test in the base field $\mathbb{F}_p$ at the cost of k/d additional cheap multiplications and applications of the Frobenius map.

We implemented the SwiftEC construction for a subset of compatible parameters in Table 1. We restricted the implementation for the cases where the JumpDivsteps approach for computing the Legendre symbol improved on the previous state-of-the-art, which means all primes except Curve25519. Coincidentally, those were the exact parameters where the corresponding elliptic curve was such that $a = 0$, which represents a simpler and faster special case of SwiftEC. Algorithm 8 presents the algorithm, which executes one inversion, two Legendre symbol computations, one square root and other minor operations in the finite field (additions, squarings and multiplications). For a fully constant-time implementation, we would need to remove the branch to handle the case in line 8, which occurs with at most negligible probability. This can be easily handled by setting P to the point at infinity, conditionally randomizing w and making the assignment in line 22 conditional as well. We omit these details to simplify the description, with a remark that most protocols would reject the point at infinity as an output anyways, requiring another hashing attempt.

Table 2 has the resulting timings for hashing to $E(\mathbb{F}_p)$, using the same benchmarking machine and iterations as described in the previous subsection. In the table, the first line represents the baseline for comparison established by RELIC's implementation of the SWU approach [WB19] in constant-time to save square root computations. This implementation computes modular square roots using exponentiation by $\frac{p+1}{4}$, relying on the fact that all benchmarked parameters have $p \equiv 3 \pmod 4$. In the second line, we have a naive implementation of the SwiftEC algorithm using Legendre symbols computed through Euler's criterion implemented in constant time using exponentiation. In the last line, we present numbers for the same implementation, but now using our algorithm for faster evaluation of the Legendre symbol using JumpDivsteps. In comparison to the constant-time baseline in the first line, SwiftEC+JumpDivstep improves timings for hashing by 51.2% to 72.4%, depending on the parameter. In comparison to the SwiftEC algorithm implemented with baseline Legendre symbols, the speedups are reduced to the range between 34.1% and 48.1%.

Table 3 has the timings for the curves supporting efficient pairing computation, now considering the cost of hashing to the twist $E'(\mathbb{F}_{p^{k/d}})$ that represents group $\mathbb{G}_2$ in the pairing setting. The first line again represents the comparison baseline from RELIC, but we note that the library did not implement efficient hashing for the larger parameters. The bottom two lines present our timings, and again we can observe that SwiftEC algorithm implemented with our constant-time algorithm for the Legendre symbol improves on the baseline implementation using exponentiation by 14.7% to 23.0%. The speedup decreases with larger parameters because multiplying by the corresponding larger cofactors dominate the total cost more prominently with higher extensions.

As an illustration of performance improvement in BLS signatures, hashing to the group $\mathbb{G}_1$ using SwiftEC+Exponentiation takes approximately the same cost as a scalar multiplication in constant time in the same group. Hence, the speedups for hashing with SwiftEC+JumpDivsteps get halved when the whole signing operation is considered. We empirically validated that observation and benchmarked BLS signing within RELIC, with and without our faster Legendre symbol evaluation. We saw that speedups ranged from 17% to 20%, which is approximately half of the speedups observed for hashing alone. The speedups for BLS verification are less than 5%, given that the pairing computation dominates. While hashing in constant time is not typically required for BLS signatures, having an efficient constant-time function is desirable from a defense-in-depth perspective [WB19].

We also integrated our JumpDivsteps implementations in the CTIDH code base [BBC+21] as a prototype. CTIDH uses the Elligator encoding [BHKL13] to sample independent points on an elliptic curve within each isogeny evaluation. We observed speedups in this routine ranging from 4.5 to 46.5, for prime moduli ranging from 511 to 2048 bits, which is quite impressive. However, the speedup was significantly reduced to only 3.5–13.5% in the overall protocol.

5 CONCLUSION

In this paper, we generalized the efficient and constant-time Bernstein-Yang (BY) algorithm for modular inversion to compute the Kronecker symbol. We defined two versions of the algorithm, with different trade-offs: a full-precision version that is easier to implement, and a faster word-oriented variant. After proving correctness of the introduced algorithms using a formalization of the BY theory within the Coq proof assistant, we produced an optimized

Table 2: Benchmarks of different approaches for hashing to $E(\mathbb{F}_p)$ for different choices of p. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime.

	secp256k1		BLS12-381		BLS24-509		BLS48-575	
	gcc	clang	gcc	clang	gcc	clang	gcc	clang
Constant-time SWU+Exp.	313,436	316,910	564,923	580,388	1,655,964	1,667,995	2,223,345	2,228,028
SwiftEC+Exponentiation	153,686	156,475	419,337	418,317	913,768	928,714	1,184,687	1,179,103
SwiftEC+JumpDivsteps	**95,442**	97,313	**275,563**	275,866	**497,852**	500,447	614,892	**614,368**

Table 3: Benchmarks of different approaches for hashing to $E'(\mathbb{F}_{p^{k/d}})$ for different choices of field. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime.

	BLS12-381		BLS24-509		BLS48-575	
	gcc	clang	gcc	clang	gcc	clang
Constant-time SWU+Exp.	1,673,040	1,734,385	-	-	-	-
SwiftEC+Exponentiation	1,246,376	1,258,655	7,519,125	7,522,370	32,065,312	32,013,390
SwiftEC+JumpDivsteps	**967,147**	976,796	**5,787,556**	5,843,306	27,461,182	**27,317,145**

Algorithm 8: Special case of SwiftEC with $a = 0$

Input : $m \in \{0,1\}^*$, elliptic curve $E(\mathbb{F}_p) : y^2 = x^3 + b$ with order $n = hr$, for prime r and cofactor h, integer λ such that $\lambda^2 \equiv -3 \pmod{p}$, hash functions $h_1, h_2 : \{0,1\}^* \to \mathbb{F}_p$, $h_s : \{0,1\}^* \to \{0,1\}$

Output: Point $P \in E(\mathbb{F}_p)$ of order r

Note : All conditional copies must be implemented in constant time with bit operations.

$t \leftarrow h_1(m), u \leftarrow h_2(m), s \leftarrow h_s(m)$
$x_1 \leftarrow (u^3 + b - t^2)$
$y_1 \leftarrow 2t^2 + x_1, z_1 \leftarrow u\lambda$
$x_1 \leftarrow x_1 z_1$
$z_1 \leftarrow 2tz_1$
$v \leftarrow z_1(x_1 - uy_1), w \leftarrow 2y_1 z_1$
if $w = 0$ **then**
  $P \leftarrow \infty$ // point at infinity
else
  $w \leftarrow w^{-1}$
  $x_1 \leftarrow vw$
  $x_2 \leftarrow -(u + x_1)$
  $x_3 \leftarrow (4y_1^2 w)^2 + u$
  $y_1 \leftarrow x_1^3 + b, y_2 \leftarrow x_2^3 + b, y_3 \leftarrow x_3^3 + b$
  $c_1 \leftarrow (y_1 + s) \bmod 2, c_2 \leftarrow \left(\frac{y_2}{p}\right)_L, c_3 \leftarrow \left(\frac{y_3}{p}\right)_L$
  $x_1 \leftarrow x_2$ **if** $c_2 = 1$ // conditional copies
  $y_1 \leftarrow y_2$ **if** $c_2 = 1$
  $x_1 \leftarrow x_3$ **if** $c_3 = 1$
  $y_1 \leftarrow y_3$ **if** $c_3 = 1$
  $y_1 \leftarrow \sqrt{y_1}$
  $y_1 \leftrightarrow -y_1$ **if** $c_1 = 1$ // choose y-coord
  $P \leftarrow h \cdot E(x_1, y_1)$ // multiply by cofactor
return P

implementation of the algorithms inside the RELIC library. Benchmarking our implementation revealed that the new algorithm is up to 40 times faster than the conventional exponentiation approach, and 25.7% faster than the previous state of the art; and improved a recent approach for hashing to elliptic curves by approximately 14.7% – 48.1%. These results show that our proposed algorithms can substantially accelerate computations in curve-based cryptography. While mapping to elliptic curves is used for illustration, any other evaluation of the Legendre or Jacobi symbols involving a secret numerator operand can perform more efficiently by using our techniques.

ACKNOWLEDGEMENTS

We would like to thank Mike Hamburg, Pieter Wuille and Tim Ruffing for discussions about their related work, and Gustavo Banegas for the help with the CTIDH code base. We also thank Yifan Zheng, Aleksei Vambol, Francisco Rodríguez-Henríquez, Gora Adj and Décio Luiz Gazzoni Filho for reporting some annoying typos in the algorithms, Daniel J. Bernstein for reporting issues with benchmarking variable-time GMP code; and the anonymous reviewers for the suggested improvements. Part of this research was supported by the Concordium Foundation.

REFERENCES

[AFK+12] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing pairings at the 192-bit security level. In *Pairing*, volume 7708 of *Lecture Notes in Computer Science*, pages 177–195. Springer, 2012.

[AGB20] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. From A to Z: projective coordinates leakage in the wild. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):428–453, 2020.

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[AGTB19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019.

[AR14] Gora Adj and Francisco Rodríguez-Henríquez. Square root computation over even extension fields. *IEEE Trans. Computers*, 63(11):2829–2841, 2014.

[ASS17] Alejandro Cabrera Aldaya, Alejandro Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended euclidean algorithm. *J. Cryptogr. Eng.*, 7(4):273–285, 2017.

[BBC+21] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. CTIDH: faster constant-time CSIDH. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):351–387, 2021.

[BCI+10] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010.

[BD19] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *J. Cryptol.*, 32(4):1298–1336, 2019.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *CCS*, pages 967–980. ACM, 2013.

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, volume 2576 of *LNCS*, pages 257–267. Springer, 2002.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[Bos14] Joppe W. Bos. Constant time modular inversion. *J. Cryptogr. Eng.*, 4(4):275–281, 2014.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.

[CCJR22] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: sublinear vélu quantum-resistant isogeny action with low exponents. *J. Cryptogr. Eng.*, 12(3):349–368, 2022.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In *ASIACRYPT (3)*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018.

[CRT22] Jorge Chávez-Saab, Francisco Rodríguez-Henríquez, and Mehdi Tibouchi. Swiftec: Shallue-van de woestijne indifferentiable function to elliptic curves - faster indifferentiable hashing to elliptic curves. In *ASIACRYPT (1)*, volume 13791 of *Lecture Notes in Computer Science*, pages 63–92. Springer, 2022.

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *S& P*, pages 1202–1219. IEEE, 2019.

[FHSS+21] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A Wood. Hashing to elliptic curves. https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/, 2021.

[Gt20] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/, 2020.

[Ham21] Mike Hamburg. Computing the Jacobi symbol using Bernstein-Yang. Cryptology ePrint Archive, Paper 2021/1271, 2021. https://eprint.iacr.org/2021/1271.

[HAS23] Benjamin Salling Hvass, Diego F. Aranha, and Bas Spitters. High-assurance field inversion for curve-based cryptography. In *CSF*, pages 552–567. IEEE, 2023.

[MAF20] Narcisse Bang Mbang, Diego F. Aranha, and Emmanuel Fouotsa. Computing the optimal ate pairing over elliptic curves with embedding degrees 54 and 48 at the 256-bit security level. *Int. J. Appl. Cryptogr.*, 4(1):45–59, 2020.

[McL21] Michael B. McLoughlin. addchain: Cryptographic addition chain generation in go. https://github.com/mmcloughlin/addchain, October 2021.

[Möl19] Niels Möller. Efficient computation of the jacobi symbol. *CoRR*, abs/1907.07795, 2019.

[MSS16] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, volume 10311 of *LNCS*, pages 83–108. Springer, 2016.

[NS22] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)mersenne prime order fields. *Adv. Math. Commun.*, 16(2):303–348, 2022.

[Por20a] Thomas Pornin. Faster modular inversion and legendre symbol, and an x25519 speed record. https://research.nccgroup.com/2020/09/28/faster-modular-inversion-and-legendre-symbol-and-an-x25519-speed-record/, 2020.

[Por20b] Thomas Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020. https://eprint.iacr.org/2020/972.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, pages 1697–1702. IEEE, 2017.

[Ros95] Harvey E Rose. *A course in number theory*. Oxford University Press, 1995.

[Sup] Supranational. The `blst` multilingual BLS12-381 signature library. https://github.com/supranational/blst.

[VR20] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the dragonfly handshake of WPA3 and eap-pwd. In *IEEE Symposium on Security and Privacy*, pages 517–533. IEEE, 2020.

[WB19] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):154–179, 2019.

[WMO21] Peter Wuille, Gregory Maxwell, and Russell O'Connor. Bounds on divsteps iterations in safegcd. https://github.com/sipa/safegcd-bounds, 2021.