IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 3, pp. 493–515. DOI:10.46586/tches.v2025.i3.493-515

Accelerating EdDSA Signature Verification with Faster Scalar Size Halving

Muhammad ElSheikh[1,2], İrem Keskinkurt Paksoy[1], Murat Cenk[3] and M. Anwar Hasan[1]

[1] University of Waterloo, Waterloo, Canada,
{mhgaelsh,ikeskink,ahasan}@uwaterloo.ca
[2] National Institute of Standards, Giza, Egypt,
[3] Ripple Labs Inc., San Francisco, USA mcenk@ripple.com

Abstract. This paper establishes that the extended Euclidean algorithm (EEA) implemented in a division-free manner is faster than the Lagrange algorithm with a similar level of optimization when it comes to halving the size of scalars found in the equations of elliptic curve signature verification. Our implementation results show that our EEA based method achieves roughly 4x speed-up for generating half-size scalars used in EdDSA. For the first time ever, EEA generated half-size scalars are used for verification of individual Ed25519 signatures yielding timing results that outperform ed25519-donna, a highly optimized open source implementation, by 16.12%. We also propose a new randomization method applied with half-size scalars to batch verification of Ed25519 signatures for which we report speed-ups compared to the well-known Bernstein et al. method for batch sizes larger than six, specifically, our method achieves 11.60% improvement for batch size 64.

Keywords: Batch Verification · EdDSA · Scalars · Signature · Verification

1 Introduction

Elliptic curve digital signatures are widely used in many applications. They have become a popular standard in applications due to their smaller key sizes and better performance compared to other types of digital signatures, such as RSA [LV01]. The efficient key sizes of elliptic curve cryptography (ECC) result in faster computations, decreased storage requirements, and reduced bandwidth usage, all of which are vital factors in many applications [KMV00]. One of the most commonly used elliptic curve signature schemes in practical applications is the Edwards-curve Digital Signature Algorithm (EdDSA) [BDL+11, BDLO12, JL17].

Generating an EdDSA digital signature involves one scalar multiplication, a computationally expensive operation. Verifying a signature, however, requires two scalar multiplications, which can become resource-intensive when many signatures need to be verified. To optimize this problem, a method called batch verification can be used. This method combines the signatures, allowing multiple signatures to be verified at the same time. This significantly reduces the computational work compared to verifying each signature one by one. This method improves the efficiency of many applications. For example, in a blockchain system, when processing transactions, the system verifies signatures to ensure the authenticity and integrity of the transactions. Many transactions are sent to the network at the same time, causing increased demands on the system's performance. Instead of checking each signature one by one, the system can check signatures using batch verification. This helps the system work faster and more efficiently. Using this method,

Received: 2025-01-15 Accepted: 2025-03-15 Published: 2025-06-05

blockchain networks can handle more transactions, reduce delays, and provide users with a better experience, especially in busy transaction environments.

In [NMVR95], a batch verification method was introduced to verify a batch of signatures using a technique that involves verifying a random linear combination of verification equations. Then, efficient approaches for batch verification of signatures and modular exponentiation were presented in [BGR98]. In [KDR+12], several algorithms for batch verification of ECDSA signatures were introduced. Bernstein et al. [BDLO12, BDL+11] developed a significantly improved method using random linear combinations, as described in [NMVR95], which utilizes an advanced scalar multiplication technique introduced by Bos-Coster. This combination leads to improved efficiency in batch verification. To enhance the performance of individual verification, fast simultaneous scalar multiplication algorithms are typically employed, which compute the sum of two elliptic curve scalar multiplications $kP + \ell Q$. Shamir's trick [ElG84, BHLM01], interleaving exponentiation [Möl01], efficient precomputation methods [OS03], and optimized computation in Montgomery form [Aki01] are examples of efficient techniques for multiple scalar multiplication. For more comprehensive details of these algorithms, we refer to Section 3.8 of the book by Hankerson et al. [HVM04].

The aforementioned approaches use random linear combinations for batch verification and advanced double scalar multiplication techniques for individual verification. On the other hand, another approach focuses on reducing the size of scalars in scalar multiplication. The article by Antipa et al. [ABG+06] uses the extended Euclidean algorithm (EEA) to reduce the size of scalars that arise in elliptic curve point multiplications in the individual verification of ECDSA signatures. The authors of the article also provide a generalization of their idea and note that to reduce the size of t (>1) scalars simultaneously, one can use $(t+1)$-dimensional lattice basis reduction techniques. For their high-speed verification of EdDSA signatures, Bernstein et al. in [BDL+11] report that the EEA based method of [ABG+06] is, however, of little benefit due to the computational overhead associated with the Euclidean computation. Following the Bernstein et al. report, Pornin's work in [Por20b] proposes an optimization of Lagrange's algorithm (LA) for lattice basis reduction in dimension 2 as an alternative to EEA to obtain half-size scalars but does not provide an optimization for EEA.

In this paper, we use a division-free EEA where we determine approximate, as opposed to exact, quotients that are powers of two. We customize the division-free EEA to generate half-size scalars and compare it with the optimized LA of [Por20b] in terms of the salient operations and execution times of the algorithms. Our results show that the new algorithm is significantly faster, e.g., it achieves roughly 4x speed-up for scalars that are about 256 bits long.

Our second contribution is the fast verification of individual EdDSA signatures using the optimized EEA mentioned above. As noted earlier, Antipa et al. [ABG+06] use EEA to speed-up individual verification of signatures of a non-standard variant of ECDSA, but to our knowledge no previous work has used it to speed-up individual verification of EdDSA signatures. For a 2^b security level, our method takes the $2b$-bit scalar associated with the public key in the verification equation and reduces it to two b-bit scalars, allowing us then to use a standard multi-scalar multiplication algorithm that takes only about b (rather than $2b$) point doubling operations. We compare the new method with the state-of-the-art implemented in ed25519-donna [Moo], a highly optimized open-source software. Our implementation results show that the new method outperforms its counterpart in ed25519-donna by 16.12% for individual verification of Ed25519 signatures.

Our third contribution relates to the batch verification of n EdDSA signatures. To our knowledge, no previous work has successfully used the technique of half-size scalars to speed up batch verification. Using our optimized EEA, we first convert each of the n verification equations into an equation that has randomization and half-size scalars associated with

non-fixed or non-base elliptic curve points. Then we aggregate the converted n equations into one and apply a well-known multi-scalar multiplication algorithm. Our implementation results for batch verification of Ed25519 signatures show that when the batch size is larger than six, our method is faster than that of Bernstein et al. [BDLO12]. We observe 11.60% improvement for 64 signatures when verified as a batch with our method.

Our final contribution is a software suite with implementations of the proposed optimized EEA for 253- and 446-bit scalars corresponding to Ed25519 and Ed448 parameters, and the new individual and batch verification methods for Ed25519 signatures. These implementations are available online at https://github.com/mhgharieb/Faster-edDSA-Verification-hEEA_q.

The remainder of this paper is organized as follows. We start by presenting the preliminary concepts in Section 2. Then, we present our division-free EEA, optimize it further for faster scalar size halving, and give a comparative analysis. Next, we delve into the accelerated EdDSA signature verification, emphasizing the improvements achieved by incorporating half-size scalars into the verification process. After that, we present our implementation results, showing the performance gains compared to state-of-the-art methods, particularly in batch verification scenarios. Finally, we conclude with a summary of our findings and their implications.

2 Preliminaries

In this section, we share some preliminary material and notation used in this paper. We also recall some related work in the literature to provide the necessary background for our work and results.

Notation. The base point (generator) of the elliptic curve (EC) group used for the signature scheme and the point at infinity will be represented by B and $\mathcal{O}$, respectively. We use the notation $\text{sign}(k)$ to indicate the sign of the integer k. The bit length of an integer k will be denoted by $\text{len}(k)$ and to avoid confusion, we assume that $\text{len}(k)$ represents the number of bits of the absolute value of k.

Classical Extended Euclidean Algorithm. Euclidean algorithm (EA) is used to determine the greatest common divisor $\gcd(\ell, v)$ of two integers ℓ and v. The extended version of EA, which is commonly referred to as extended Euclidean algorithm (EEA), outputs a pair of integers s and t, called Bézout coefficients for (ℓ, v), along with $\gcd(\ell, v)$ that satisfy $\gcd(\ell, v) = s\ell + tv$. An iterative version of the classical EEA is given in Algorithm 1.

Algorithm 1 EEA(ℓ, v)

Input: Two integers ℓ, v, $\ell > v$
Output: The greatest common divisor $r = \gcd(\ell, v)$ of ℓ and v, and two integers s, t such that $r = s\ell + tv$

1: $r_{-1} \leftarrow \ell$, $r_0 \leftarrow v$, $s_{-1} \leftarrow 1$, $s_0 \leftarrow 0$, $t_{-1} \leftarrow 0$, $t_0 \leftarrow 1$, $i \leftarrow 0$
2: **while** $r_i \neq 0$ **do**
3: $\quad i \leftarrow i + 1$
4: $\quad q_i \leftarrow \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor$
5: $\quad r_i \leftarrow r_{i-2} - q_i r_{i-1}$
6: $\quad s_i \leftarrow s_{i-2} - q_i s_{i-1}$
7: $\quad t_i \leftarrow t_{i-2} - q_i t_{i-1}$
8: **end while**
9: **return** $r = r_{i-1}$, $s = s_{i-1}$, $t = t_{i-1}$

In [ABG+06], Antipa et al. used EEA to obtain half-size scalars to accelerate ECDSA verification. Their technique applies EEA until the bit length of r_i and t_i become nearly half-size of the bit length of ℓ, where ℓ represents the EC group order. More explicitly, the technique in [ABG+06] uses Algorithm 1 with the condition $r_i > \sqrt{\ell}$ in the second step instead of $r_i \neq 0$ and ignores the sixth step. We refer to this version as the half extended Euclidean algorithm and denote it by hEEA. The division operation in EEA is often a slow process and there are many studies in the literature that use a division-free version of the classical EEA to avoid the computational overhead of division, like [BY19] and [Has01], to name a few. In Section 3.2.2, we present our version of division-free hEEA, that enhances the performance of obtaining half-size scalars for EdDSA verification.

Lattice Basis Reduction and Reduced Size Scalars. In [ABG+06], the authors suggest using lattice basis reduction as another way to reduce scalar sizes. To this end, Pornin's work in [Por20b] uses Lagrange's algorithm (LA), where the author first provides an improved version of the classical LA by making it division-free and then optimizing it further to generate reduced-size scalars for Schnorr and EdDSA signature verification. The optimized method is presented below as Algorithm 2.

Algorithm 2 Optimized Lagrange's reduce_basis algorithm [Por20b]

Input: Prime number ℓ, integer v such that $0 \leq v < \ell$
Output: A short non-zero vector (ρ, τ) such that $\rho \equiv \tau v \mod \ell$

1: $N_0 = \ell^2$, $N_1 = v^2 + 1$, $p = \ell v$
2: $s \leftarrow \lfloor (\text{len}(\ell) + 4)/2 \rfloor$, $stop\_value \leftarrow \text{len}(\ell) + 1$
3: $(r_0, t_0) \leftarrow (\ell, 0)(\mod \pm 2^s)$
4: $(r_1, t_1) \leftarrow (v, 1)(\mod \pm 2^s)$
5: **while** $\text{len}(N_1) > stop\_value$ **do**
6: $\quad d \leftarrow \max(0, \text{len}(p) - \text{len}(N_1))$
7: $\quad$ **if** $p > 0$ **then**
8: $\quad\quad (r_0, t_0) \leftarrow (r_0 - 2^d r_1, t_0 - 2^d t_1)$
9: $\quad\quad N_0 \leftarrow N_0 + 2^{2d} N_1 - 2^{d+1} p$
10: $\quad\quad p \leftarrow p - 2^d N_1$
11: $\quad$ **else**
12: $\quad\quad (r_0, t_0) \leftarrow (r_0 + 2^d r_1, t_0 + 2^d t_1)$
13: $\quad\quad N_0 \leftarrow N_0 + 2^{2d} N_1 + 2^{d+1} p$
14: $\quad\quad p \leftarrow p + 2^d N_1$
15: $\quad$ **end if**
16: $\quad$ **if** $N_0 < N_1$ **then**
17: $\quad\quad (r_0, t_0) \leftrightarrow (r_1, t_1)$
18: $\quad\quad N_0 \leftrightarrow N_1$
19: $\quad$ **end if**
20: **end while**
21: **return** $(\rho, \tau) = (r_1, t_1)$

The reduce_basis algorithm given above keeps track of seven variables N_0, N_1, p, r_0, r_1, t_0, and t_1. The variables N_0, N_1, and p are initialized as integers of at most $2\text{len}(\ell)$ bits in length. The initial values of the variables r_0 and r_1 have $\lfloor (\text{len}(\ell) + 4)/2 \rfloor$ bit each, whereas the initial values of t_0 and t_1 have only one bit.

EdDSA Verification. EdDSA [JL17] is defined on twisted Edwards curves [BBJ+08] over a prime field $\mathbb{F}_p$, and produces Schnorr-like signatures [Sch91]. We refer the reader to [JL17] for certain details that are omitted here, such as the encoding/decoding of

elements, and provide a simplified version of the verification algorithm. Let $\sigma(R, s)$ be an EdDSA signature on a message M with a public key A. The verifier, upon obtaining $(\sigma(R, s), A, M)$, first computes a hash value h using R, A, and M, then accepts the signature σ if $sB = R + hA$ holds and rejects otherwise.

Batch Verification of EdDSA Signatures. Batch cryptography was first introduced by Shamir in [Fia90] for RSA. In [NMVR95], the first batch verification algorithm for Digital Signature Algorithm (DSA) was proposed. Since then, numerous other studies have contributed to the literature such as [KDR+12, BDLO12, BGR98]. We recall the definition of batch verification from [BDLO12] for EdDSA signatures.

Let $\Sigma_n = \{(\sigma_i = (R_i, s_i), A_i, M_i) | i = 1, \ldots, n\}$ be a batch of n signatures produced by EdDSA. To verify the signatures in Σ_n as a batch, using randomizers is essential as explained in [BDLO12]. Randomized batch verification of n signatures proposed by Bernstein et al. requires sampling b-bit random numbers z_i for each signature. Multiplying each $s_i B \stackrel{?}{=} R_i + h_i A_i$ with the corresponding randomizer z_i, adding them to each other and rearranging the terms gives the following:

$$\left(\sum_{i=1}^{n} z_i s_i\right) B - \sum_{i=1}^{n} z_i R_i - \sum_{i=1}^{n} (z_i h_i) A_i \stackrel{?}{=} \mathcal{O}, \tag{1}$$

where all scalar operations (shown in parentheses) are done modulo the group order ℓ. If all the signatures in the batch are valid, then (1) holds. A common way to check whether Equation (1) holds is to use a multiscalar multiplication (MSM) algorithm such as Bos-Coster [BC90, de 95]. These algorithms are more efficient than the classical multiply-add method.

3 EEA Based Half-size Scalars

Antipa et al. [ABG+06] improved the verification of a modified version of the ECDSA signature using half-size scalars determined by EEA explained in Section 2. However, the overhead due to the Euclidean computation can be relatively excessive in some situations [BDL+11]. To this end, in this section we first present a division-free EEA and then optimize it for faster scalar size halving.

3.1 Division-free Extended Euclidean Algorithm

Here we present an improved version of the classical EEA, in which we eliminate the exact division in Step 4 of Algorithm 1 by employing an approximate value for q_i, that is, a power of two. We refer to our version as EEA_approx_q. In addition, each quotient being a power of two replaces multiplications in Algorithm 1 with simple shift operations. Unlike the classical version, here the main variables can, however, be negative, forcing us to determine the sign of the approximate remainder in each iteration. The pseudocode of our improved algorithm is given in Algorithm 3 followed by a correctness analysis.

Algorithm 3 tracks values of two variables r, two s and two t. These variables are present in the algorithm for the sake of completeness and correctness analysis. To obtain half-size scalars, we do not need the variables s. An implementation-friendly pseudocode for obtaining half-size scalars is given later in this section.

Proposition 1. *Algorithm 3 terminates.*

Proof. Each iteration i of Algorithm 3 has an output r_i that is obtained from the inputs r_{i-2} and r_{i-1}, such that $r_i = r_{i-2} \pm 2^d r_{i-1}$ (see lines 5-10 of Algorithm 3) which immediately implies $\text{len}(r_i) \leq m_i$, for $m_i = \max\{\text{len}(r_{i-2}), \text{len}(r_{i-1})\}$. We prove the theorem by

Algorithm 3 EEA_approx_q(ℓ, v)

Input: Two integers ℓ and v with $\ell > v$
Output: Three integers r, s and t such that $r = s\ell + tv$ and $|r| = \gcd(\ell, v)$

$r_{-1} \leftarrow \ell$, $r_0 \leftarrow v$, $s_{-1} \leftarrow 1$, $s_0 \leftarrow 0$, $t_{-1} \leftarrow 0$, $t_0 \leftarrow 1$, $i \leftarrow 0$, $lenr_0 \leftarrow \text{len}(\ell)$, $lenr_1 \leftarrow \text{len}(v)$
while $r_i \neq 0$ **do**
  $i \leftarrow i + 1$
  $d \leftarrow \max\{0, lenr_0 - lenr_1\}$
  if $(\text{sign}(r_{i-2}) = \text{sign}(r_{i-1}))$ **then**
    $q_i \leftarrow 2^d$
  else
    $q_i \leftarrow -2^d$,
  end if
  $r_i \leftarrow r_{i-2} - q_i r_{i-1}$
  $s_i \leftarrow s_{i-2} - q_i s_{i-1}$
  $t_i \leftarrow t_{i-2} - q_i t_{i-1}$
  $lenr_0 \leftarrow lenr_1$, $lenr_1 \leftarrow \text{len}(r_i)$
end while
return $r = r_{i-1}$, $s = s_{i-1}$, $t = t_{i-1}$

showing that the sequence $m_0,\ m_1,\ m_2,\ \ldots$, which we denote by $\{m_i\}$, is monotonically decreasing.

Depending on the lengths of the inputs r_{i-2} and r_{i-1}, there are three cases to consider for each iteration.

$$\begin{aligned}
\textbf{Case I: } &\text{len}(r_{i-2}) > \text{len}(r_{i-1}) \implies \text{len}(r_i) < \text{len}(r_{i-2}) \text{ and } m_i < m_{i-1} \\
\textbf{Case II: } &\text{len}(r_{i-2}) = \text{len}(r_{i-1}) \implies \text{len}(r_i) < \text{len}(r_{i-1}) \text{ and } m_i = m_{i-1} \\
\textbf{Case III: } &\text{len}(r_{i-2}) < \text{len}(r_{i-1}) \implies \text{len}(r_i) \leq \text{len}(r_{i-1}) \text{ and } m_i = m_{i-1}
\end{aligned}$$

When an iteration's inputs satisfy Case I, it is guaranteed that m_i is strictly less than m_{i-1}. If the inputs of iteration i satisfy Case II, then we have $m_i = m_{i-1}$ and the next iteration falls into Case I, which means $m_{i+1} < m_i$. Similarly, if we have Case III for i-th iteration, then either iteration $i + 1$ or $i + 2$ (when $i + 1$ satisfies Case II) must fall into Case I. Therefore, the sequence $\{m_i\}$ is monotonically decreasing. □

Proposition 2. *Algorithm 3 produces* $\pm\gcd(\ell, v)$ *and, for any iteration* i*, satisfies*

$$s_i\ell + t_i v = r_i, \tag{2}$$

A proof is given in Appendix C.

Multiplicative inverse. It is easy to see that for an integer v such that $0 < v < \ell$, where ℓ and v are co-prime, the multiplicative inverse of v is obtained from the output of Algorithm 3 as follows.

$$v^{-1} \equiv r_{f-1}t_{f-1} \mod \ell \tag{3}$$

where r_{f-1} and t_{f-1} are two return values when the algorithm finishes after f iterations. For ℓ being a prime, $r_{f-1} \in \{-1, 1\}$, implying that the cost of Equation (3) is a modular reduction incurred only when r_{f-1} and t_{f-1} are of opposite sign and the reduction in this case is simply an add/sub operation.

3.2 Generating half-size scalars

3.2.1 Underlying rationale

Similar to EEA (Algorithm 1), which has been shown to *heuristically* compute two half-size scalars in [ABG+06], Algorithm 3 is used here to produce half-size scalars by changing its stop condition in the while loop (line 2). For this, we first present the following proposition for Algorithm 3. The expression in the proposition also holds for Algorithm 1. A proof of the proposition is in Appendix C.

Proposition 3. *For any iteration i, Algorithm 3 satisfies*

$$t_i r_{i-1} - t_{i-1} r_i = (-1)^i \ell. \tag{4}$$

Bit-lengths increase/decrease. In Algorithm 3, when the inputs ℓ and v are co-prime and the algorithm finishes after f iterations, $r_f = 0$, $|r_{f-1}| = \gcd(\ell, v) = 1$ and then for $i = f$, Proposition 3 yields $|t_f| = \ell$. Although not necessarily monotonically, we show that the length of the values of variable r progressively decreases from $\text{len}(\ell)$ at initialization to 0 at the end (see the proof of Proposition 1). Our experiments indicate that the sum $\text{len}(r_i) + \text{len}(t_i) \approx \text{len}(\ell)$ for $i \leq f$. Based on this empirical observation, we can say that the length of the values of variable t gradually increase from 0 at initialization to $\text{len}(\ell)$ at the end.

Stopping half-way for bit-length balance. Suppose that the stop condition of the while loop in Algorithm 3 is changed to $\text{len}(r_i) > \frac{1}{2}\text{len}(\ell)$, which is roughly half the way to the full gcd computation. With the new stop condition, assume that the algorithm ends after $e \leq f$ iterations, i.e., $\text{len}(r_e) \leq \frac{1}{2}\text{len}(\ell)$, but $\text{len}(r_i) > \frac{1}{2}\text{len}(\ell)$ for $i < e$. Our experiments with large values of ℓ and randomly chosen $v \in [1, \ell - 1]$ indicate that on average $\text{len}(r_e)$ is slightly less than $\frac{1}{2}\text{len}(\ell)$ and so is $\text{len}(t_e)$. Table 3 in Section 5 presents our experimental results.

Effect. Substituting the half-size values r_e and t_e into Equation (2) yields the following.

$$r_e \equiv t_e v \mod \ell \tag{5}$$

A key consequence of Equation (5) is that in multiscalar multiplication a $\text{len}(\ell)$-bit scalar can be converted to a half-size scalar, reducing the number of point-doubling operations. This is elaborated in Section 4 in the context of EdDSA signature verification.

3.2.2 Faster Scalar Size Halving

To generate half size scalars, we incorporate the aforementioned half-way stopping to Algorithm 3. We also employ a tweak to the algorithm that reduces the number of iterations. The tweak is explained in the following.

Based on the proof of Proposition 1, the sequence $\{m_i\}$ decreases at least one in any three consecutive iterations, i.e., $m_{i+2} < m_{i-1}$. The tweak is to ensure that $\text{len}(r_{i-2}) \geq \text{len}(r_{i-1})$ for each i, which eliminates Case III mentioned in the above proof. This means that at least one of every two consecutive iterations satisfies Case I. Therefore, $\{m_i\}$ decreases at least one in any two iterations, i.e., $m_{i+2} < m_i$, thus making the tweaked algorithm terminates with fewer iterations. The tweak essentially replaces line 13 of Algorithm 3 with a set of statements realizing the following pseudocode so that $lenr_0 \geq lenr_1$ is maintained in each iteration.

if $\text{len}(r_i) > \text{len}(r_{i-1})$ **then**
  $lenr_0 \leftarrow \text{len}(r_i), r_i \leftrightarrow r_{i-1}, s_i \leftrightarrow s_{i-1}, t_i \leftrightarrow t_{i-1}$

else
$lenr_0 \leftarrow lenr_1, lenr_1 \leftarrow \text{len}(r_i)$
end if

Algorithm 4 shown below generates half size scalars and is obtained from Algorithm 3 by i) changing the stop condition as discussed above, ii) removing s variables as they are not needed for scalar generation, iii) dropping variable subscripts that are associated with the loop index to reduce memory requirement, iv) applying the tweak mentioned above, and v) describing the variable updating in a way that simplifies the task of software coding. Our experiments with approximately 253-bit long inputs show that the tweak reduces the average number of iterations by roughly 23%.

Algorithm 4 hEEA_approx_q(ℓ, v)

Input: Prime number ℓ, integer v such that $0 < v < \ell$
Output: Two half-size integers (ρ, τ) such that $\rho \equiv \tau v \mod \ell$

$r_0 \leftarrow \ell$, $r_1 \leftarrow v$, $t_0 \leftarrow 0$, $t_1 \leftarrow 1$, $stop\_value \approx \frac{\text{len}(\ell)}{2}$, $lenr_0 \leftarrow \text{len}(\ell)$, $lenr_1 \leftarrow \text{len}(v)$
while $lenr_1 > stop\_value$ **do**
  $d = lenr_0 - lenr_1$
  if $(\text{sign}(r_0) = \text{sign}(r_1))$ **then**
    $r \leftarrow r_0 - (r_1 \ll d)$
    $t \leftarrow t_0 - (t_1 \ll d)$
  else
    $r \leftarrow r_0 + (r_1 \ll d)$
    $t \leftarrow t_0 + (t_1 \ll d)$
  end if
  if $\text{len}(r) > lenr_1$ **then**
    $r_0 \leftarrow r$, $t_0 \leftarrow t$
    $lenr_0 \leftarrow \text{len}(r)$
  else
    $r_0 \leftarrow r_1$, $r_1 \leftarrow r$, $t_0 \leftarrow t_1$, $t_1 \leftarrow t$
    $lenr_0 \leftarrow lenr_1$, $lenr_1 \leftarrow \text{len}(r)$
  end if
end while
return $\rho = r_1, \tau = t_1$

Number of iterations: Since in any two consecutive iterations the bit length of one of the two variables r is reduced by at least one and the while loop stop condition is $lenr_1 > \text{len}(\ell)/2$, the maximum number of iterations is $\approx 2(\text{len}(\ell) - \frac{1}{2}\text{len}(\ell) + \text{len}(v) - \frac{1}{2}\text{len}(\ell)) = 2\text{len}(v) \leq \text{len}(\ell) + \text{len}(v) \leq 2\text{len}(\ell)$, assuming that $\text{len}(v)$ is greater than the stop condition. We ran two experiments to observe the average number of iterations of Algorithm 4 for the values of ℓ that are the orders of the EC groups defined in Ed25519 and Ed448-Goldilocks [Ham15]. For the ℓ value in each of the curves, we executed Algorithm 4 for more than 50 million times with randomly chosen v values that satisfy $0 < v < \ell$. Our results show that the average number of iterations for the value of ℓ in Ed25519 is approximately 95 with a standard deviation of 6.4, and that for Ed448 is 169 with a standard deviation of 8.5.

Computation: In each iteration, one instance of bit length determination occurs (i.e., $\text{len}(r)$). As discussed above, with the progress of the algorithm, the length of r_1 decreases from about $\text{len}(\ell)$ to $\text{len}(\ell)/2$. In terms of the main arithmetic operations, in each iteration the algorithm performs one add/sub involving the variables r and another add/sub involving the variables t. As mentioned above, the length of the variables t increases from 0 to about $\text{len}(\ell)/2$. We also note that the add/subs are just regular integer operations, not modular

Table 1: Comparison of scalar size halving algorithms

Features	Algorithm 2 [Por20b]	Algorithm 4
# Half-size variables*	4	2
# Full-size variables	$0^\ddagger$	2
# Double-size variables	3	0
Max # of iterations	$4 \times \text{len}(\ell)$	$2 \times \text{len}(\ell)$
# Length determination per iteration	1 or 2	1
# Half-size add/sub per iteration	2	1
# Full-size add/sub per iteration	$0^\ddagger$	1
# Double-size add/sub per iteration	3	0
# Shift operations per iteration	5	2
# Double-size comparator per iteration	1	0

* For both algorithms, half-, full- and double-sizes correspond to $\text{len}(\ell)/2$, $\text{len}(\ell)$ and $2\text{len}(\ell)$, respectively.
‡ Assuming that full-size variables are stored and updated only up to the half-size [Por20b].

arithmetic.

On the basis of the aforementioned features, we can compare the EEA based Algorithm 4 with the Optimized Lagrange lattice basis reduction algorithm proposed in [Por20b] and given as Algorithm 2. Both algorithms determine half-size scalars and are division-free. Compared to Algorithm 4 and in a nutshell, Algorithm 2 requires i) more variables, some of which are twice the size of the algorithm inputs, ii) has a higher value for the maximum number of iterations, and iii) requires more computation. In Table 1, we contrast the two algorithms. In Section 5.1, we provide the results of our software implementation for these two algorithms for input sizes geared towards Ed25519 and Ed448.

Remark 1. While the well-known binary extended GCD, see Chapter 14 of [MvV97] for a description, is an efficient method for determining gcd and is an analogue of EEA in terms of producing gcd of two integers along with their Bézout coefficients, it in its present known form does not appear to produce half-size scalars. In the binary extended GCD algorithm, when one of its primary variables, analogous to variables r in EEA, is divided by 2, the corresponding secondary variables, analogous to variables t in EEA, is added, if not even, with ℓ, making the size of the variable to increase to roughly that of ℓ.

4 Accelerated EdDSA Signature Verification

In this section, we present an improvement for the individual verification of EdDSA signatures by using hEEA_approx_q. Moreover, we propose a new randomization method for batch verification of EdDSA signatures. In this section, we use ℓ to denote the prime order of the subgroup used in EdDSA, and we assume that $\text{len}(\ell) = 2b$.

4.1 Faster Individual Verification

Formulation facilitating half-size scalars. Suppose $(\sigma = (R, s), A, M)$ is a signature produced by EdDSA and the order ℓ of the elliptic curve group is known. To verify this signature, first we compute the hash value h using R, A, and M, and then check whether (6) holds or not.

$$sB \stackrel{?}{=} R + hA \Longleftrightarrow R \stackrel{?}{=} sB - hA. \tag{6}$$

Similar to the methods in [ABG+06, Por20b], our method makes use of two half-size integers. By calling hEEA_approx_q$(\ell, v = h)$, we produce two half-size integers ρ and τ

such that $\rho \equiv \tau h \mod \ell$. Once we obtain the half-size scalars, we multiply the verification equation in (6) by τ. Then we have the following:

$$\begin{aligned} \tau sB &\stackrel{?}{=} \tau R + \tau hA, \\ \tau sB &\stackrel{?}{=} \tau R + \rho A. \end{aligned} \tag{7}$$

The value of τs in (7) is a $2b$-bit integer. Therefore, we can express it as $\lambda_1 + 2^b\lambda_2$ for two b-bit integers λ_1 and λ_2. Since B is a system parameter, $B' = 2^bB$ can be stored as a precomputed value.

$$\lambda_1 B + \lambda_2 B' - \tau R - \rho A \stackrel{?}{=} \mathcal{O} \tag{8}$$

Therefore, we transform the verification equation from (7) into (8), in which the LHS can be computed via a quadruple-scalar multiplication algorithm[1]. Finally, the LHS is multiplied by the cofactor of the EdDSA curve before checking whether it is equal to $\mathcal{O}$.

Cost analysis. Straus' algorithm [Str64] is known for fast multi-scalar multiplication involved in verification of EdDSA signatures. Here we apply the algorithm to the quadruple scalar multiplication (QSM) in Equation (8). For window width c, it involves roughly b point doublings, four point additions following every c doublings, $3(2^c-2)$ point additions as part of initial computation of jA, jR and jB', where $j \in [2, 2^c-1]$ and free precomputation of the corresponding jB. This results in approximately $b + \frac{4}{c}(1 - \frac{1}{2^c})b + 3(2^c - 2)$ point additions/doublings and 2^c-2 precomputed points. Since B' is fixed, rather than including jB' as part of the initial computation, in addition to jB one can also precompute jB', in which case the number of precomputed points is doubled and increases to $2 \times (2^c - 2) = (2^{c+1} - 4)$ but the initial computation reduces to $2(2^c - 2)$. As a result, transitioning from a single precomputation table for jB to two precomputation tables, one for jB and the other for jB', reduces the total cost by $2^c - 2$ point additions.

The authors of [BDLO12] also apply the Straus algorithm to EdDSA signature verification using the original form shown in Equation (6) that involves double scalar multiplication (DSM) with roughly $2b$ bit scalars, and report a cost of $2b + \frac{2}{c}(1 - \frac{1}{2^c})2b + (2^c - 2)$ point additions/doublings and $2^c - 2$ precomputed points for jB. Doubling the size of the precomputation table for jB increases the number of precomputed points to $(2^{c+1} - 2)$, and in every $c+1$ doublings, on average one addition is expected that involves a point from the precomputation table to perform the DSM of Equation (6). This reduces the average number of additions involving points stored in the table from $\frac{2b}{c}(1 - \frac{1}{2^c})$ to $\frac{2b}{c+1}(1 - \frac{1}{2^{c+1}})$, i.e., lowering the total cost by roughly $\frac{2b}{c(c+1)}$ additions.

When it comes to comparing the cost of signature verification using Equation (6)/DSM vs. Equation (8)/QSM and relying on the same number of precomputed points, i.e., $2^c - 2$, the latter requires approximately $b - 2(2^c - 2)$ fewer additions/doublings of points, but has the overhead of decoding the compressed R as well as making a call to hEEA_approx_q. Furthermore, doubling the number of percomputed points reduces the total cost by $(2^c - 2)$ point additions in the case of QSM, compared to roughly $\frac{2b}{c(c+1)}$ point additions in the case of DSM.

In order to obtain further insight, we have conducted experiments using the python scripts[2] provided by Bernstein et al. [BDLO12], which uses Straus' method with two speedup techniques, namely signed digits and sliding windows. Our results are shown in Table 2 averaging data from 100,000 random samples. Our implementation results in terms of clock cycles for another set of experiments using ed25519-donna, which uses a low-level programming language, are presented in Section 5.

[1] Given that the two half-size integers τ, ρ are signed, Equation (8) is adjusted based on their signs to maintain the half-size scalar multiplication property.

[2] The scripts `separate3.py` and `dobulescalarmult.py` are available on https://cr.yp.to/badbatch.html

Table 2: Cost comparison for multi-scalar multiplications involved in individual verification using Straus' algorithm

Verification technique	Precomputation Table	# Point add/double
Equation (6)/DSM	none*	$2.78b^\ddagger$
	jB	$2.72b$
Equation (8)/QSM	none	$1.92b$
	jB	$1.85b$
	jB and jB'	$1.79b$

* After carefully reviewing the code, we figured out that the table for jB is computed on-the-fly similar to jA.
‡ Consistent with $2.8b$ reported in [BDLO12].

Remark 2. By viewing Equation (8) as two double-scalar multiplications and employing two processor cores, a verifier can reduce the *latency* of the task. One core computes $\lambda_1 B - \rho A$ and the other $\lambda_2 B' - \tau R$, each using Straus' method with $2^c - 2$ points precomputed offline and $b + \frac{2}{c}(1 - \frac{1}{2^c})b + 2^c - 2$ point additions/doublings performed online. Latency reduction is important for some applications, but is not pursued in this work.

4.2 Batch verification with hEEA_approx_q

New randomization technique. The standard technique for secure batch verification is to multiply each individual verification equation with a sufficiently large random number, aggregate them into one and then check whether the latter holds. For EdDSA the randomized i-th equation as suggested in [BDL+12] is

$$z_i s_i B \stackrel{?}{=} z_i R_i + z_i h_i A_i, \tag{9}$$

where z_i is a b-bit random number for 2^b security level. Thus the right-hand side of Equation (9) has two scalar multiplications, one involving a half-size scalar and the other a full-size. An EEA based scalar size halving technique, e.g., Algorithm 4, applied in a straightforward way, is not effective in reducing the size of both scalars simultaneously. As seen in the case of individual verification in Section 4.1, Algorithm 4 works well in limiting the maximum scalar size on the right-hand side of a verification equation to b bits when one of the scalars is equal to unity. Based on this observation, we randomize the i-th individual verification equation by multiplying h_i with the inverse of a random number picked for the batch and then apply Algorithm 4 to generate two half-size scalars. When we aggregate all the randomized equations, this procedure results in two multi-scalar multiplications on the right-hand side with half-size scalars. Below, we describe the procedure step by step.

1. Choose $2b$-bit random nonzero integer $U < \ell$, and compute U' such that $UU' \equiv 1 \mod \ell$.

2. Call hEEA_approx_q with inputs ℓ and $v_i = h_i U' \mod \ell$ to obtain two b-bit integers ρ_i and τ_i such that $\rho_i \equiv \tau_i v_i \mod \ell$ or equivalently $\tau_i h_i \equiv U \rho_i \mod \ell$.

3. Multiply the i-th equation by τ_i for each i :

$$\tau_i s_i B \stackrel{?}{=} \tau_i R_i + U \rho_i A_i. \tag{10}$$

4. Sum up all the equations in (10):

$$\underbrace{\left(\sum \tau_i s_i\right)}_{s} B \stackrel{?}{=} \underbrace{\left(\sum \tau_i R_i\right)}_{R} + U \underbrace{\left(\sum \rho_i A_i\right)}_{A}, \tag{11}$$

   obtain s, and compute R and A using a multi-scalar-multiplication (MSM) algorithm, resulting in $sB \stackrel{?}{=} R + UA$.

5. Call hEEA_approx_q again with inputs ℓ and U to obtain b-bit integers ρ and τ satisfying $\rho \equiv \tau U \mod \ell$, and multiply $sB \stackrel{?}{=} R + UA$ by τ:

$$\tau s B \stackrel{?}{=} \tau R + \rho A. \tag{12}$$

6. Since Equation (12) is similar to Equation (7) in the previous section, the cofactored Equation (8) is used for the final verification.

Cost. Batch verification of n signatures using our aforementioned method requires two MSMs with n scalars of b bits each, whereas the same task using the existing method based on Equation (1) requires one MSM involving $n + 1$ scalars of size $2b$ bits each and another n scalars of b bits each. For batch verification, the efficiency of MSM algorithms, e.g., the Bos-Coster method that costs $O(nb/\lg n)$ point operations, is affected more by the size of the scalars than the quantity of them. Therefore, our method improves the batch verification process by reducing the size of the scalars rather than reducing the number of scalar multiplications.

Implementation notes. We applied our new batch technique on Ed25519. The benchmark results of our C implementation can be seen in Section 5.2. We also note that among the computations in Step 4 of our aforementioned batch verification method, the two MSMs involving non-fixed points are the most demanding. The two MSMs are, however, independent and balanced in terms of computational load, opening the door for parallel computing using two processor cores running concurrently and requiring little communication between them and hence to potentially reduce the latency of the overall process. In the classical method, such parallelization is also possible, but the computational load is not balanced, since one MSM involves $2b$-bit scalars and the other involves b-bit scalars. For the new method, it is also possible to parallelize the computation in Step 6 using two cores and it is similar to the case of individual verification discussed in Section 4.1. Such latency reduction is however not further explored in this work.

Same signer. Equation (11) is mainly for separate signers. For batch verification involving a single signer with public key A^*, Equation (11) becomes

$$\left(\sum \tau_i s_i\right) B \stackrel{?}{=} \left(\sum \tau_i R_i\right) + \left(U \sum \rho_i\right) A^*. \tag{13}$$

When no additional optimization is applied, the cost of Equation (13) is one MSM involving b-bit scalars and two scalar multiplications each involving $2b$-bit scalars. This is the same as the cost of the existing batch verification method based on Equation (1) for a single signer. As a result, batch verification for a single signer, Equation (1) tailored for that signer is computationally less expensive as it avoids the calling of the hEEA_approx_q algorithm to generate half size scalars stated in Step 2 of the new method. However, after performing the MSM in the single-signer specific version of Equation (1), the hEEA_approx_q algorithm should be called once to generate two half size scalars as in Step 5.

Other notes. We can rearrange the classical batch verification equation in Equation (9) as $z_i P_i \stackrel{?}{=} \mathcal{O}$, where $P_i = R_i + h_i A_i - s_i B$. Consequently, the probability that an invalid signature i (i.e., $P_i \neq \mathcal{O}$) can pass batch verification (i.e., $z_i P_i = \mathcal{O}$) is at most 2^{-b} for a b-bit random z_i [BDLO12].

Similarly, our batch verification equation in Equation (10), which can be rearranged as $\tau_i P_i \stackrel{?}{=} \mathcal{O}$. Therefore, the probability that an invalid random signature i (i.e., $P_i \neq \mathcal{O}$) can pass batch verification (i.e., $\tau_i P_i = \mathcal{O}$) is at most $2^{-b'} \gtrapprox 2^{-b}$, where $b' = \text{avg}(\text{len}(\tau_i))$. Since our hEEA_approx_q algorithm produces τ_i and ρ_i values that are nearly $b = \text{len}(\ell)/2$ bits long (but less than b on average), where ℓ is the larger input of the algorithm, we can adjust the size of the output values by replacing ℓ with $L = \ell \cdot w$, for a sufficiently large integer w.

For example, if we want τ_i and ρ_i to be nearly b bits long, it would suffice to provide Algorithm 4 with positive integers $L = \ell \cdot w$ and $v_i < L$, where w is chosen such that $\text{len}(w) = b - b'$. By definition, τ_i and ρ_i satisfy $\tau_i v_i \equiv \rho_i \mod L$, which implies $\tau_i v_i \equiv \rho_i \mod \ell$ since $L = \ell \cdot w$.

Another method of regulating the bit length of the outputs ρ and τ, is to adjust the value of *stop_value* in the algorithm. hEEA_approx_q terminates when $\text{len}(\rho) \leq$ *stop_value*, so $\text{len}(\rho)$ decreases (resp. increases) as *stop_value* decreases (resp. increases). Since $\text{len}(\tau)$ is inversely related to $\text{len}(\rho)$, to increase the bit length of τ, we can decrease *stop_value*.

5 Implementation Results

In this section, we provide our x86 (64-bit) architecture implementation and benchmarking for hEEA_approx_q for Ed25519 and Ed448, as well as EdDSA verification for Ed25519. Additionally, in Appendix A we explore the use of EEA_approx_q to obtain multiplicative inverses modulo the prime $p = 2^{255} - 19$, allowing us to understand the performance of EEA_approx_q relative to state-of-the-art constant-time gcd algorithms developed for the same purpose. All experiments were performed using Intel i7-9750H(Coffee Lake) @ 2.60GHz (TurboBoost is disabled).

The C implementation is available as an open-source GitHub repository:
https://github.com/mhgharieb/Faster-edDSA-Verification-hEEA_q.

5.1 Scalar Size Halving Implementation

hEEA_approx_q. Our hEEA_approx_q (Algorithm 4) and Pornin's scheme [Por20b], referred to as reduce_basis, (Algorithm 2) have many common features; specifically, how the variables are updated. In order to compare these two algorithms in a fair way, we follow the same integer representation used in Pornin's implementation of reduce_basis, for the implementation of Algorithm 4. In other words, integer values are represented as sequences of limbs, where each limb is 64 bits long and fits into a 64-bit type (`uint64_t`).

For Ed25519, since $\text{len}(\ell) = 253$ and $\lceil \text{len}(\ell)/2 \rceil = 127$, both r_0 and r_1 (with the extra sign bits) can fit into 4 limbs each, and both t_0 and t_1 (with the extra sign bits) fit into 2 limbs each. For Ed448, $\text{len}(\ell) = 446$ and $\lceil \text{len}(\ell)/2 \rceil = 223$, so 7 limbs are suitable for each r_0, r_1, and 4 limbs for each t_0 and t_1.

Since $|r_0|$ and $|r_1|$ decrease as Algorithm 4 progresses, performance can be improved by switching their representations to ones with fewer limbs when their values, including the extra sign bits, become small enough. In Ed25519, we switch to 3-limb representations for r_0 and r_1 when $lenr_0$ and $lenr_1$ are less than 192 (191 bits for absolute values and 1 bit for the sign). For Ed448, we switch to 6, 5, and 4 limbs when $lenr_0$ and $lenr_1$ are less than 384, 320, and 256, respectively.

Additions and subtractions are performed on a limb-by-limb basis. Specifically, we employed the carry/borrow flag offered by the CPU. For example, we used the intrinsic

functions `_addcarry_u64` and `_subborrow_u64` to perform 64-bit addition and subtraction with carry on Intel processors.

Algorithm 4 relies on evaluating the exact bit lengths of r_0 and r_1 (similar to Algorithm 2) to determine the left shift value d and when to terminate. We followed the same approach proposed in [Por20b] to determine these bit lengths. First, the most significant non-zero limb is located, followed by the most significant non-zero bit within that limb. The latter step can be performed by determining the number of leading zeros in that limb, which can be efficiently computed on the x86 (64-bit) architecture. In particular, we used the `_lzcnt_u64()` intrinsic function to determine the Leading Zero Count for 64-bit integers.

Using Classical EEA. To obtain half-size scalars using the classical EEA, we employ the GMP library [GNUa] to implement the division-based hEEA, which is the same EEA(Algorithm 1) with ignoring the sixth step and the stop condition of the while loop became $(\text{len}(r_i) > stop\_value)$, similar to `hEEA_approx_q`.

Using HGCD. The HGCD (Half GCD) function is a generalization to Lehmer's algorithm [Leh38] to accelerate the computation of the GCD for large input sizes. The variant of Lehmer's algorithm implemented in GMP library [GNUa, GNUb] is `mpn_hgcd2` function.

Let the inputs a and b each have N limbs, and let $S = \lfloor N/2 \rfloor + 1$. The internal GMP function `mpn_hgcd`(a, b) returns a transformation matrix M with non-negative elements and reduced numbers $(\alpha; \beta) = M^{-1}(a; b)$. The reduced numbers α and β are constrained such that their sizes are greater than S limbs, while their absolute difference $|\alpha - \beta|$ must fit within S limbs. The matrix elements are also approximately $N/2$ limbs in size [GNUb].

Given that $\det(M) = 1$, the matrices M and its inverse can be represented as

$$M = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}, \quad M^{-1} = \begin{bmatrix} u_{11} & -u_{01} \\ -u_{10} & u_{00} \end{bmatrix},$$

where $\det(M)$ denotes the determinant of the matrix M. Thus, the reduced numbers α and β can be expressed as

$$\alpha = \quad u_{11} \cdot a - u_{01} \cdot b \tag{14}$$

$$\beta = -u_{10} \cdot a + u_{00} \cdot b \tag{15}$$

Here, the sizes of α and β are $\gtrapprox S$ limbs, while u_{01} and u_{00} are each roughly $N/2$ limbs in size.

Recall that during the running of the Extended Euclidean Algorithm (EEA) with the inputs a and b, for the i-th iteration, we have $r_i = s_i \cdot a + t_i \cdot b$. Therefore, Equations (14) and (15) are equivalent to two iterations of the EEA.

To obtain two half-sized integers (ρ, τ) such that $\rho \equiv \tau v \pmod{\ell}$, we first call `mpn_hgcd` with inputs (ℓ, v) to compute the outputs α and β. Subsequently, we apply several iterations of the classical EEA until the desired half-size is achieved, as outlined in Algorithm 5.

Benchmark. Our implementation for this part includes eight C files, as outlined below, where `*` represents either `curve25519` or `curve448`:

- '`*_hEEA_vartime.c`': Implementations of Algorithm 4.
- '`*_reduce_basis_vartime.c`': Modified versions of the implementation of Algorithm 2 for Curve25519 and Curve448, adapted from the Curve9767 implementation provided in [Por20b].
- '`*_hEEA_div_vartime.c`': Implementations of the division-based hEEA using GMP library to obtain half-size scalars.

Algorithm 5 **hSIZE_HGCD**(ℓ, v)

Input: Prime number ℓ, integer v such that $0 < v < \ell$
Output: Two half-size integers (ρ, τ) such that $\rho \equiv \tau v \mod \ell$

1: $a \leftarrow \ell$, $b \leftarrow v$
2: $(\alpha, \beta, M) \leftarrow$ `mpn_hgcd`(a, b)
3: $r_0 \leftarrow \alpha$, $r_1 \leftarrow \beta$, $t_0 \leftarrow -u_{01}$, $t_1 \leftarrow u_{00}$, $stop\_value \approx \frac{\text{len}(\ell)}{2}$
4: **if** $r_0 < r_1$ **then**
5: $\quad r_0 \leftrightarrow r_1$, $t_0 \leftrightarrow t_1$
6: **end if**
7: **while** $\text{len}(r_1) > stop\_value$ **do**
8: $\quad q \leftarrow \left\lfloor \frac{r_0}{r_1} \right\rfloor$
9: $\quad r \leftarrow r_0 - qr_1$
10: $\quad r_0 \leftarrow r_1, r_1 \leftarrow r$
11: $\quad t \leftarrow t_0 - qt_1$
12: $\quad t_0 \leftarrow t_1$
13: $\quad t_1 \leftarrow t$
14: **end while**
15: **return** $\rho = r_1, \tau = t_1$

`mpz_tdiv_qr`(q, r_0, r_0, r_1)
`mpz_swap`(r_0, r_1)
`mpz_submul`(t_0, q, t_1)
`mpz_swap`(t_0, t_1)

Table 3: The average number of iterations and the bit lengths of ρ and τ, where mean and standard deviation are written as 'Mean(STD)'

	reduce_basis[Por20b]			hEEA_approx_q		
	# of Iteration	Bit length of ρ	Bit length of τ	# of Iteration	Bit length of ρ	Bit length of τ
Ed22519	94.95(6.26)	125.37(1.55)	124.36(1.54)	95.16(6.39)	125.85(1.49)	123.80(1.48)
Ed448	168.91(8.31)	222.37(1.55)	221.36(1.54)	169.71(8.51)	221.85(1.49)	221.80(1.48)

- '`*_hgcd_vartime.c`': Implementations of Algorithm 5 using GMP `mpn_hgcd` function.

These files are compiled using the Clang-10.0.0 compiler with flags: `-O3 -mlzcnt`.

Table 3 presents statistics for reduce_basis and hEEA_approx_q based on 50 million random values of v, showing the average number of iterations and the bit lengths of ρ and τ.

Moreover, in Table 4, we report the execution times of hEEA_approx_q, reduce_basis, hEEA, and hSIZE_HGCD, measured in clock cycles, averaged over 10,000 random instances of v. For each instance v, the clock cycles are measured as the average of 10 repeated calls to the algorithm being tested with the same input value v. As shown in Table 4, our hEEA_approx_q implementation for Ed25519 achieved a speedup of 3.91, 8.64, and 5.07 compared to reduce_basis, hEEA, and hSIZE_HGCD, respectively. These correspond to performance improvements of 74.45%, 88.43%, and 80.31%, respectively. For Ed448, our hEEA_approx_q implementation achieved a speedup of 4.43, 5.71, and 1.39, respectively.

As can be seen in Table 4, we observe an inconsistency in the performance of hSIZE_HGCD between Ed25519 and Ed448, compared with the other algorithms. In Appendix B, we investigate the reasons behind this inconsistency and propose two techniques to improve the performance of hSIZE_HGCD for Ed25519.

5.2 Ed25519 Signature Verification

ed25519-donna [Moo] is a fast and highly optimized 32-bit and 64-bit implementation of Ed25519, written in assembly language. It is based on optimizations from the amd64-64-

Table 4: Cycle counts of algorithms for producing half-size scalars and improvement percentages of hEEA_approx_q

	hEEA_approx_q	reduce_basis	hEEA‡	hSIZE_HGCD‡
Ed25519	3531.47	13823.46 (74.45%)	30514.30 (88.43%)	17933.06 (80.31%)
Ed448	9899.21	43884.44 (77.44%)	56525.29 (82.49%)	13760.03 (28.06%)

‡ Implemented using GMP library [GNUa], version 6.2.0 [GNUb].

24k and amd64-51-30k implementations by the authors of Curve25519. ed25519-donna demonstrates high performance in both individual and batch verifications. Therefore, we use it as a performance baseline. For a fair comparison, we updated ed25519-donna to support our new approaches for both individual and batch verifications in addition to the classical methods, and then performed benchmarking[3].

Individual Verification. In ed25519-donna, an individual verification, as described in Equation (6), is performed as follows: use double-scalar multiplication to compute $sB - hA$, and then check whether the encoding of the result matches the encoding of R, skipping the decoding of compressed R. The double-scalar multiplication is optimized by using the wNAF (Windowed Non-Adjacent Form) representation of the two full-size 253-bit scalars, along with a precomputation table for jB with the required odd values of j. We call this version DSM_B. We also implement a version of the double-scalar multiplication in which the size of the precomputation table for jB is doubled, and call this version as DSM_B_doublePre.

Our approach to individual verification, as described in Equation (8), requires a linear combination of four points: B and B' are fixed, while R and A depend on the signature, with four half-size, approximately 127-bit, scalars (λ_1, λ_2, τ, and ρ). One approach to compute it is by using two double-scalar multiplications but with half-size scalars. A faster approach is to employ quadruple-scalar multiplication using the same wNAF representation as ed25519-donna's double-scalar multiplication, with two precomputation tables for jB and jB' with the required odd values of j (the size of each table is the same as the jB table used in DSM_B). This quadruple-scalar multiplication, referred to as QSM_B_B', is faster than two double-scalar multiplications. To make a fair comparison, we also implement the quadruple-scalar multiplication with a single precomputation table for jB, similar to DSM_B, and call this version as QSM_B.

The overhead of our approach arises from decoding the compressed R, handling hEEA_approx_q, and performing quadruple-scalar multiplication with four half-size scalars involving two fixed points and two variable points, compared to ed25519-donna's double-scalar multiplication with two full-size scalars involving one fixed point and one variable point.

Batch Verification. In ed25519-donna, the dominant part of batch verification is the multi-scalar multiplication in Equation (1), which is performed using the Bos-Coster algorithm [BC90, de 95]. For a batch size of n, there are $2n + 1$ scalars: $n + 1$ full-size scalars and n half-size scalars.

In our approach, as described in Section 4.2, we first compute U', the multiplicative inverse of U. Since this computation is performed once per batch verification, we use the GMP library [GNUa] for simplicity. However, performance could be improved by precomputing multiple values for the pair (U, U') and passing a pair with the batch, or by

[3] ed25519-donna implementation is cofactorless, i.e., it does not multiply Equation (6) by the cofactor. For fairness, our implementation also omits multiplying Equation (8) by the cofactor.

Table 5: Performance improvement of our approach using QSM with hEEA_approx_q over classical ed25519-donna using DSM for individual verification

DSM-based approach		Our QSM-based approach		Speed up	Improvement
DSM_B	157752.02	QSM_B	132327.36	1.1921	16.12 %
DSM_B_doublePre	156586.85	QSM_B_B'	125044.72	1.2522	20.14 %

employing a faster method for computing the multiplicative inverse, such as our Algorithm 3 or [BY19, Por20a].

For a batch size of n, we execute hEEA_approx_q algorithm n times to obtain the half-size ρ_i and τ_i for each signature. As described in Equation (11), we require two multi-scalar multiplications with n half-size scalars each to compute $R = \sum \tau_i R_i$ and $A = \sum \rho_i A_i$. We adapt ed25519-donna's Bos-Coster implementation to handle n half-size scalars. Finally, we perform an equivalent individual verification, as described in the previous section, to check the result in Equation (12).

Benchmark. Our implementation adds three C files to ed25519-donna as follows:

- `new_batch_helper.h`: Contains several helper functions for performing individual and batch verification, including the multiplicative inverse function, a function to reformat the output of the `curve25519_hEEA_vartime` function to the format used in ed25519-donna, and the double-scalar multiplication DSM_B_doublePre as well as the two versions, QSM_B_B' and QSM_B, of the quadruple-scalar multiplication function.

- `ed25519-donna-open_new.h`: Implements the DSM-based individual verification method using DSM_B_doublePre, as well as the two versions of the new QSM-based individual verification method, one using QSM_B_B' and the other using QSM_B.

- `ed25519-donna-batchverify_new.h`: Implements the new batch verification method.

The files were compiled using the GCC-9.4.0 compiler with flags: `-O3 -m64`.

In Table 5, we report the execution time, measured in clock cycles, for individual verification, averaged over the 10,000 random signatures. For each signature verification, cycles are measured as the average of 10 repeated calls to the same verification function with the same signature. As shown in Table 5, our QSM-based individual verification implementation with QSM_B outperforms ed25519-donna with DSM_B by 16.12%, which is equivalent to a speedup of 1.19. Furthermore, using QSM_B_B' for individual verification results in an improvement of 20.14% over QSM_B_doublePre and a speedup of 1.25. This confirms that the performance gain by transitioning from QSM_B to QSM_B_B' is greater than that from DSM_B to DSM_B_doublePre as shown in the cost analysis in Section 4.1.

Figure 1 depicts the execution time of the new and classical i.e., ed25519-donna batch verification, measured in clock cycles per verification and averaged over 100 random batch signatures, where the batch size varies from 4 to 128 signatures per batch. For each batch signature, the batch verification function is called 10 times and the clock cycles are measured in the same way as in the individual verification. Figure 2 shows the improvement of batch verification over the new individual verification method. Our implementation results show that starting from a batch size of seven, the performance of our new approach exceeds the classical one by 3.33%. Table 6 summarizes the performance improvement of the new approach over the classical one for batch sizes ranging from 4 to 128, where the new approach achieves up to 12.39% improvement in performance.

Table 6: Performance improvement of our approach over classical ed25519-donna for batch verification

Batch size	ed25519-donna[Moo]	Our approach	Speed up	Improvement
4	131145.15	143320.68	0.9150	-9.28 %
8	106204.45	102196.24	1.0392	3.77 %
16	90376.79	82709.25	1.0927	8.48 %
32	80660.46	71612.68	1.1263	11.22 %
64	74182.59	65576.06	1.1312	11.60 %
128	70412.98	61687.25	1.1415	12.39 %

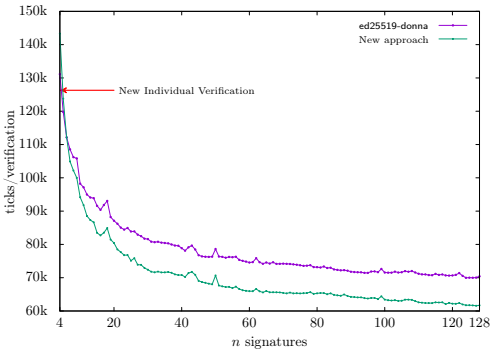

Figure 1: Execution Time of batch verification in cycles/verification.

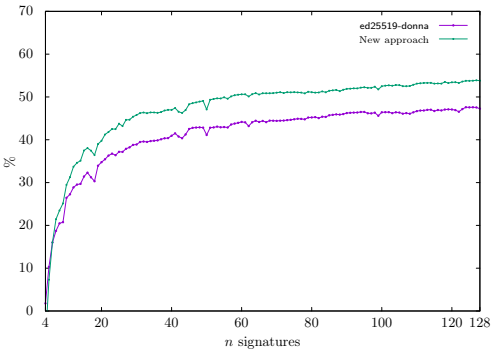

Figure 2: Improvement of using batch verification over our new individual verification.

6 Concluding Remarks

In this paper, we have presented an optimization of EEA to speed-up the halving the size of scalars found in the verification of elliptic curve signatures. Whether individual or batch, the verification process can be meaningfully sped up using half-size scalars. In this paper, we have applied such scalars to the verification of EdDSA signatures yielding improved timings results. We are currently applying them to ECDSA and expect to report our results in an upcoming publication. Making the algorithm run in constant time is important for some applications and is worth investigating. As sketched earlier, our verification methods, individual as well as batch, offer opportunities to reduce the latency of the overall process using multiple cores. This can be further explored.

References

[ABG+06] Adrian Antipa, Daniel R. L. Brown, Robert Gallant, Rob Lambert, René Struik, and Scott A. Vanstone. Accelerated verification of ECDSA signatures. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 307–318. Springer, Heidelberg, August 2006.

[Aki01] Toru Akishita. Fast simultaneous scalar multiplication on elliptic curve with Montgomery form. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 255–267. Springer, Heidelberg, August 2001.

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *AFRICACRYPT 08*, volume 5023 of *LNCS*, pages 389–405. Springer, Heidelberg, June 2008.

[BC90] Jurjen N. Bos and Matthijs J. Coster. Addition chain heuristics. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 400–407. Springer, Heidelberg, August 1990.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, Heidelberg, September / October 2011.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 454–473. Springer, Heidelberg, December 2012.

[BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 236–250. Springer, Heidelberg, May / June 1998.

[BHLM01] Michael Brown, Darrel Hankerson, Julio Cesar López-Hernández, and Alfred Menezes. Software implementation of the NIST elliptic curves over prime fields. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 250–265. Springer, Heidelberg, April 2001.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298.

[de 95] Peter de Rooij. Efficient exponentiation using procomputation and vector addition chains. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 389–399. Springer, Heidelberg, May 1995.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, August 1984.

[Fia90] Amos Fiat. Batch RSA. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 175–185. Springer, Heidelberg, August 1990.

[GNUa] GNU Project. GNU Multiple Precision Arithmetic Library. https://gmplib.org/. Accessed: 2024-10-09.

[GNUb] GNU Project. The GNU Multiple Precision Arithmetic Library Manual. https://gmplib.org/gmp-man-6.2.0.pdf. Accessed: 2025-01-01.

[Ham15] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. https://eprint.iacr.org/2015/625.

[Has01] Mohammed Anwarul Hasan. Efficient computation of multiplicative inverses for cryptographic applications. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pages 66–72. IEEE, 2001.

[HVM04] Darrel Hankerson, Scott Vanstone, and Alfred Menezes. *Guide to Elliptic Curve Cryptography*. Springer New York, NY, 1 edition, 2004.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. https://www.rfc-editor.org/info/rfc8032.

[KDR+12] Sabyasachi Karati, Abhijit Das, Dipanwita Roychowdhury, Bhargav Bellur, Debojyoti Bhattacharya, and Aravind Iyer. Batch verification of ECDSA signatures. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT 12*, volume 7374 of *LNCS*, pages 1–18. Springer, Heidelberg, July 2012.

[KMV00] Neal Koblitz, Alfred Menezes, and Scott A. Vanstone. The state of elliptic curve cryptography. *DCC*, 19(2/3):173–193, 2000.

[Leh38] Derrick H Lehmer. Euclid's Algorithm for Large Numbers. *The American Mathematical Monthly*, 45(4):227–233, 1938.

[LV01] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, September 2001.

[Möl01] Bodo Möller. Algorithms for multi-exponentiation. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 165–180. Springer, Heidelberg, August 2001.

[Moo] Andrew Moon. ed25519-donna: Implementations of a Fast Elliptic-Curve Digital Signature Algorithm. https://github.com/floodyberry/ed25519-donna. original-date: 2012-03-02T22:15:52Z, Accessed: 2024-10-09.

[MvV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[NMVR95] David Naccache, David M'Raïhi, Serge Vaudenay, and Dan Raphaeli. Can D.S.A. be improved? Complexity trade-offs with the digital signature standard. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 77–85. Springer, Heidelberg, May 1995.

[OS03] Katsuyuki Okeya and Kouichi Sakurai. Fast multi-scalar multiplication methods on elliptic curves with precomputation strategy using Montgomery trick. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 564–578. Springer, Heidelberg, August 2003.

[Por20a] Thomas Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Report 2020/972, 2020. https://eprint.iacr.org/2020/972.

[Por20b] Thomas Pornin. Optimized lattice basis reduction in dimension 2, and fast schnorr and EdDSA signature verification. Cryptology ePrint Archive, Report 2020/454, 2020. https://eprint.iacr.org/2020/454.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.

[Str64] Ernst G Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70(806-808):16, 1964.

A Modular Inversion

As shown in Section 3.1, our proposed EEA_approx_q (Algorithm 3) can be used to compute the multiplicative inverse of v from Equation (3), $v^{-1} \equiv r_{f-1}t_{f-1} \mod \ell$, where $r_{f-1} \in \{-1, 1\}$ and t_{f-1} are the two return values produced when the algorithm concludes after f iterations.

In this section, we investigate the application of EEA_approx_q to compute multiplicative inverses modulo the prime $p = 2^{255} - 19$, which is used in Curve25519. Bernstein and Yang [BY19] proposed a constant-time GCD algorithm, referred to as safeGCD, for this purpose. The performance results reported in [BY19], for safeGCD on Haswell, Skylake, and Kaby Lake, are 10,050 cycles, 8778 cycles, and 8543 cycles, respectively. Pornin [Por20a] introduced a constant-time binary GCD algorithm, referred to as binGCD, for inversion modulo $p = 2^{255} - m$ (where m is small), and compared it with safeGCD for $m = 19$. The reported results for binGCD and safeGCD from [Por20a] are 6253 and 8520 cycles on Coffee Lake, respectively. The following section presents our benchmarking results for inversion modulo $p = 2^{255} - 19$ using EEA_approx_q.

Benchmark. We implemented EEA_approx_q in the same way as our hEEA_approx_q implementation described in Section 5.1. Additionally, we obtained the source code for binGCD. However, the original source code of safeGCD used to achieve the aforementioned performance (8520 cycles) is unavailable online. As a result, we rely on the performance data reported by Pornin [Por20a], since our system also uses Coffee Lake.

Our benchmark results are averaged over 10,000 random instances of v. To maintain consistency with Pornin's methodology, for each value v, we measured clock cycles using the `rdtsc` opcode before and after a sequence of 100 dependent inversions, where each inversion's output serves as the input for the next.

We achieve an average of 5230 cycles for EEA_approx_q, while 6299 cycles for the constant-time binGCD, indicating that the former runs in 16.97% fewer cycles. The above-mentioned cycle count of 6299 is consistent with the 6253 cycles for binGCD reported in [Por20a]. Consequently, we estimate that the original version of safeGCD would run at approximately 8520 cycles, as reported in [Por20a].

It is worth noting that two improved versions of safeGCD are available, with the latest version running in 3939 cycles on our system. Since these versions are implemented in assembly code, it is challenging to determine the reasons for the significant performance improvement between the latest version and the original. In future, we plan to explore the techniques underlying these improvements and investigate their applicability to our implementation. Furthermore, we plan to explore the applicability of making EEA_approx_q run in constant time.

B Custom optimization of Algorithm 5 for Ed25519

As mentioned at the end of Section 5.1, we observe an inconsistency in the performance of hSIZE_HGCD between Ed25519 and Ed448. In this section, we investigate potential reasons for this inconsistency and propose two techniques to improve the performance of hSIZE_HGCD for Ed25519.

Inconsistency. We believe that this inconsistency arises for the following reason. GMP functions operate on a limb-based structure, and the stop condition of `mpn_hgcd` requires

the reduced numbers α and β to have more than $S = \lfloor N/2 \rfloor + 1$ limbs [GNUb]. As a result, in the case of Ed448 where $S = 4$, the bit-length of α and β is $\gtrapprox 256$, which is close to the *stop_value* = 223, located in the fourth limb. As a result, only a few EEA iterations are needed to reach the *stop_value*. Conversely, for Ed25519, $S = 3$, thus the bit-length of α and β is $\gtrapprox 192$. This requires more EEA iterations to reach the *stop_value* = 127, which is located in the second limb.

By adjusting the stop condition of `mpn_hgcd`, the performance of Algorithm 5 can be improved for Ed25519.

First Optimization. Instead of using GMP `mpn_hgcd` with $S = \lfloor N/2 \rfloor + 1$, we propose using a custom version, `mpn_hgcd_custom`, where $S = \lfloor N/2 \rfloor$. In this case, $S = 2$, and the bit-length of α and β is $\gtrapprox 128$, which is very close to the *stop_value* = 127. Our experiment shows that the number of clock cycles required by hSIZE_HGCD is reduced from 17933 to 4542 by using this trick. However, our proposed hEEA_approx_q, which requires 3531 clock cycles, is still superior to this version of hSIZE_HGCD.

Second Optimization. Recall that the reduced numbers α and β must be larger than S limbs, while their difference $abs(\alpha - \beta)$ must fit within S limbs [GNUb]. In the case of $S = 2$, the size of $abs(\alpha - \beta)$ is exactly 128 bits.

In the context of our proposed approaches for Ed25519 verifications, it is acceptable to work with 128-bit scalars instead of 127-bit scalars. Therefore, we can omit the EEA iterations after calling `mpn_hgcd_custom` and return:

$$\rho = abs(\alpha - \beta), \qquad \tau = (\alpha > \beta) \ ? \ -(u_{00} + u_{01}) : (u_{00} + u_{01})$$

By using this version of hSIZE_HGCD, the number of clock cycles is reduced to 2975, which is 556 fewer, on average, than the clock cycles required by hEEA_approx_q.

Ed25519 Verification Using the New hSIZE_HGCD. We conducted experiments to benchmark the new version of hSIZE_HGCD in our approaches for both individual and batch Ed25519 verifications. To ensure fairness, we reused the same test set and benchmarking methodology employed for testing hEEA_approx_q.

Our experiments reveal that individual verification using hSIZE_HGCD requires an average of 124630 clock cycles, compared to 125044 clock cycles with hEEA_approx_q, as shown in Table 5. This represents an average difference of 414 clock cycles, which is smaller than the difference between hSIZE_HGCD and hEEA_approx_q observed earlier. The reduced difference is due to the scalars' bit-length returned by hSIZE_HGCD, which is 128 bits, compared to an average of 125 bits for hEEA_approx_q, as indicated in Table 3. As a result, the number of doubling/addition operations during QSM is lower when using hEEA_approx_q, leading to a smaller observed difference in clock cycles. In terms of batch verification, using the new hSIZE_HGCD improves the performance with an improvement range of 1.13% to 2.16% compared to hEEA_approx_q.

C Proofs

C.1 Proof of Proposition 2

Proof. Suppose that Algorithm 3 finishes after f iterations, i.e. $r_f = 0$ but $r_i \neq 0$ for $i < f$. For any i in the algorithm, we have $\gcd(r_{i-2}, r_{i-1}) = \gcd(r_{i-1}, r_i)$, since $r_i = r_{i-2} - q_i r_{i-1}$. As the value of r_i can be negative, we have $\gcd(\ell, v) = \gcd(r_0, r_1) = \cdots = \gcd(r_{f-1}, r_f) = \gcd(r_{f-1}, 0) = |r_{f-1}|$, where $|\cdot|$ denotes the absolute value.

Now, turning to Equation (2), it is clearly true for $i = -1$ and $i = 0$. To prove it by induction, assume that it holds for $i - 2$ and $i - 1$, that is, $s_{i-2}\ell + t_{i-2}v = r_{i-2}$ and

$s_{i-1}\ell + t_{i-1}v = r_{i-1}$, respectively. Referring to lines 10-12 of Algorithm 3, we can then write

$$\begin{aligned} s_i\ell + t_i v &= (s_{i-2} - q_i s_{i-1})\ell + (t_{i-2} - q_i t_{i-1})v \\ &= s_{i-2}\ell + t_{i-2}v - q_i(s_{i-1}\ell + t_{i-1}v) \\ &= r_{i-2} - q_i r_{i-1} = r_i. \end{aligned}$$

□

C.2 Proof of Proposition 3

Proof. Equation (4) holds for $i = 0$. To prove by induction, assume that it holds for $i - 1$, i.e., $t_{i-1}r_{i-2} - t_{i-2}r_{i-1} = (-1)^{i-1}\ell$. The latter along with lines 10 and 12 of the algorithm yields:

$$\begin{aligned} t_i r_{i-1} - t_{i-1}r_i &= (t_{i-2} - q_i t_{i-1})r_{i-1} - t_{i-1}(r_{i-2} - q_i r_{i-1}) \\ &= (t_{i-2}r_{i-1} - t_{i-1}r_{i-2}) - q_i(t_{i-1}r_{i-1} - t_{i-1}r_{i-1}) \\ &= -(-1)^{i-1}\ell = (-1)^i\ell. \end{aligned}$$

□