

Scalable Small ECDLP Solver via Windowed-Batch Inversion for Additively Homomorphic Encryption

Murat Cenk^{1*} Muhammad ElSheikh^{2,3*} Irem Kesinkurt Paksoy²
M. Anwar Hasan²

¹Ripple Labs Inc., San Francisco, CA, USA. mcenk@ripple.com

²University of Waterloo, Waterloo, ON, Canada. {mhgaels, ikeskink, ahasan}@uwaterloo.ca

³National Institute of Standards, Giza, Egypt

Abstract

Decryption in EC-ElGamal and Twisted ElGamal based Additively Homomorphic Encryption (AHE) requires solving a small exponential Elliptic Curve Discrete Logarithm Problem (ECDLP) that recovers a bounded plaintext scalar m from $m \cdot G$. This recovery step is the performance-critical path in various applications, such as electronic voting systems (e-Voting), private federated learning, and privacy-preserving blockchain confidential payment systems including Zether (EC-ElGamal), Solana Token-2022 Confidential Transfers (Twisted ElGamal), and the forthcoming XRPL XLS-96 standard (EC-ElGamal). The state-of-the-art FastECDLP (Tang et al., 2023) employs the baby-step giant-step algorithm (BSGS) to solve ECDLP in ℓ -bit range by utilizing a baby-step table T'_1 of 2^{ℓ_1-1} affine points and achieves fast decryption by precomputing a giant-step table T'_2 of all 2^{ℓ_2-1} affine points where $\ell_1 + \ell_2 = \ell$. Even with the signed-range halving of Tang et al., the giant-step table for secp256k1 requires 5.91 GB for a 58-bit search space and exponentially increases to 378 GB for a 64-bit search space. Thus, as noted by Tang et al., a 64-bit range remains impractical on a typical workstation due to the memory requirements. We present three algorithmic contributions that together eliminate T'_2 and enable practical 64-bit decryption. (i) A Jacobian coordinate loop advances the giant-step point iteratively in Jacobian coordinates, removing the need to precompute the giant steps in affine coordinates. (ii) Windowed batch inversion batch-inverts a window of a small W accumulated Z -coordinates at once, reducing the per-thread working memory required for the giant-step computation from $O(2^{\ell_2})$ (the size of T'_2 in FastECDLP) to $O(W)$ regardless of bit size. (iii) Co- Z arithmetic computes two giant steps simultaneously by maintaining a shared Z -coordinate, reducing the field multiplication cost per step. We validate our solver on two machines, a laptop-class Apple M1 Pro and a desktop-class Intel Core Ultra 9 285K, with consistent results across both. On the Intel machine, our complete solver outperforms a faithful re-implementation of FastECDLP at 54-bit and 58-bit ($\ell_1 = 31$, up to $2.3\times$ faster) without storing the giant-step table T'_2 . At 63-bit and 64-bit, Tang et al.'s giant-step table T'_2 requires 189 GB and 378 GB respectively, making their approach infeasible on typical desktop or laptop computers. So far, our solver is the only feasible approach, completing full 63-bit and 64-bit decryption in 53 sec and 105 sec, directly overcoming the memory limitation identified by Tang et al. We also discuss two additional arithmetic optimizations, an inversion-chaining scheme and a co- Z Jacobian addition, that reduce the per-step arithmetic cost.

Keywords: Elliptic curve discrete logarithm, EC-ElGamal, baby-step giant-step, Jacobian coordinates, batch inversion, cuckoo hashing, blockchain, confidential transactions, homomorphic encryption, secp256k1.

1 Introduction

The growing demand for privacy-preserving technologies has increased the importance of cryptographic techniques that protect sensitive data while enabling computation. Among these techniques, additively homomorphic encryption (AHE) [13, 26] enables computations to be performed directly on ciphertexts while preserving the privacy of the underlying data. AHE therefore plays an important role in privacy-preserving

*These authors contributed equally to this work and share first authorship.

applications including electronic voting [11, 27, 36], online auctions [12, 16], machine learning [23, 20], federated learning [37, 14, 38], confidential blockchain transactions [10, 21], and secure data aggregation in wireless sensor networks (WSNs) [1, 31]. Due to their efficiency advantages and ability to provide comparable security with smaller key sizes, elliptic-curve (EC)-based AHE schemes have been employed in a range of privacy-preserving applications.

For example, public blockchains such as Ethereum [5, 32] and XRPL [34] record all transactions permanently and publicly. While pseudonymous, all movement of funds between addresses is traceable. For financial applications such as payroll, institutional transfers, this transparency is often unacceptable. Bünz et al. [4], Cenk et al. [7], and Chen et al. [10] employ an EC-ElGamal-based AHE for confidential blockchain transactions, allowing amounts to be summed without being revealed, and using zero-knowledge proofs [6, 9] to enforce correctness. Similarly, WSNs collect sensitive measurements that are typically aggregated before being processed by a central entity. However, collecting individual sensor readings may reveal sensitive information. Privacy-preserving data aggregation schemes proposed by Bahi et al. [1] and Ugus et al. [31] address this issue by employing EC-based AHE to encrypt sensor measurements, thereby allowing aggregate values to be computed without decrypting individual measurements. The fundamental cryptographic primitive underlying these EC-based AHE schemes is plaintext recovery: given $P_m = m \cdot G$, the expression obtained after decryption, recover the bounded plaintext scalar $m \in [0, 2^\ell)$.

Several major blockchain platforms have deployed or are actively developing such schemes. Zether [4] is the trustless confidential payment mechanism for Ethereum, keeping balances encrypted under EC-ElGamal with zero-knowledge range proofs. Solana Token-2022 Confidential Transfers [29] natively embeds Twisted ElGamal encryption into the token standard. To bound decryption cost, balances are split into 16-bit and 48-bit components. The XRP Ledger is developing confidential transfers under XLS-96 [35] for the Multi-Purpose Token (MPT) standard based on EC-ElGamal, where the XLS-33 specification defines amounts with maximum value $2^{63} - 1$, requiring the decryption solver to cover the full range $[0, 2^{63})$. Both EC-ElGamal and Twisted ElGamal require solving the same small ECDLP during decryption that is to recover the bounded scalar m from $m \cdot G$. Unlike the general ECDLP, which is computationally infeasible, the bounded scalar makes the problem tractable via meet-in-the-middle methods. The Baby-Step Giant-Step (BSGS) algorithm of Shanks [28] is the method of choice, achieving time $O(2^{\ell/2})$ with memory $O(2^{\ell/2})$ where ℓ is the bit length of the search range, a square-root improvement over brute force. The dominant cost is modular field inversion: each giant step requires converting an elliptic curve point to affine coordinates, costing approximately $103 \times$ a field multiplication (measured on Apple M-series, see Section 2).

The problem has received sustained attention in the cryptography community. Bernstein and Lange [3] establish foundational algorithmic improvements for small discrete logarithms. Galbraith et al. [15] exploit point addition symmetry for a constant-factor speedup in the general ECDLP setting. For the bounded plaintext case arising in AHE decryption, Chatzigiannis et al. [8] target AHE decryption specifically, proposing truncated x -coordinate lookup tables with reduced storage. The current state of the art is FastECDLP [30], which combines BSGS with tree-based Montgomery batch inversion (TreeMon) [24], $k = 3$ cuckoo hashing [25], and 64-bit key truncation, achieving 248 ms at 52-bit and 990 ms at 54-bit on an Intel Xeon at 16 threads. Following this work, Xie et al. [33] propose Fast G_T DLP, an algorithm for solving small-exponent discrete logarithms in the multiplicative group G_T arising from elliptic-curve pairings, rather than ECDLP, which is the focus of this work. A recent work by Liao and Zhang [19] proposes a new variant of BSGS for solving ECDLP using elliptic divisibility sequences (EDS). Their experimental results demonstrate improvements over conventional BSGS for the tested search ranges, including instances of up to 50 bits. In comparison, FastECDLP reports solving ECDLP instances with search spaces of up to 54 bits, which exceeds the largest search range reported for the EDS-based approach. Even though FastECDLP proposes a practical solution for small-range ECDLP, it requires precomputing a full giant-step table T'_2 in affine coordinates, which demands approximately 378 GB at 64 bits, far beyond ordinary workstation capacity. Tang et al. note that full 64-bit decryption is impractical on a typical workstation due to the memory requirements of T'_2 .

This work makes three algorithmic contributions that jointly remove the need for T'_2 and make full-range decryption practical. First, a Jacobian coordinate loop keeps the giant-step running point in Jacobian form throughout, removing the need to precompute T'_2 in affine. Second, windowed batch inversion batch-inverts a window of W accumulated Z -coordinates at once, reducing per-thread working memory from $O(2^{\ell_2})$ to

$O(W) \approx 50$ KB regardless of bit size. Third, Co- Z arithmetic reduces the field multiplication cost per iteration by simultaneously computing two giant steps while maintaining a shared Z -coordinate. As a practical engineering contribution, we additionally parallelise the denominator computation and batch inversion (Phase 1+2) across threads, reducing the sequential bottleneck in Tang et al.’s original architecture. We validate this solver on two machines, a laptop-class Apple M1 Pro and a desktop-class Intel Core Ultra 9 285K. Against a faithful re-implementation of FastECDLP on the same hardware, our complete solver is competitive or faster at 54-bit and 58-bit ($\ell_1 = 31$, up to $2.3\times$ on the Intel machine) without storing any giant-step table. The larger gain appears at 63-bit and 64-bit, where FastECDLP’s table T'_2 grows to 189 GB and 378 GB and becomes infeasible on either machine, whereas our solver completes both ranges in 53 sec and 105 sec respectively, closing the memory gap that Tang et al. identified but could not overcome.

The remainder of this paper is organized as follows. Section 2 fixes notation and reviews the machinery the construction rests on: EC-ElGamal decryption and the bounded ECDLP instance it induces, affine and Jacobian curve arithmetic together with their relative field-operation costs, Montgomery batch inversion, the BSGS algorithm, and the FastECDLP baseline whose precomputed table T'_2 is the source of the memory bottleneck we remove. Section 3 then develops our solver in three steps: advancing the giant-step point in Jacobian coordinates, batch-inverting the accumulated Z -coordinates one window at a time, and computing a forward and a backward giant step together under a shared Z -coordinate so that the signed range is covered at close to the cost of a single step. Section 4 looks beyond a single decryption, bounding the total work when several amounts must be recovered at once and showing how prior knowledge of the plaintext range or its structure shortens the search. The measurements follow in Section 5, covering the C implementations we compare, the choice of window size, and a same-hardware comparison against our re-implementation of FastECDLP on both machines, before Section 6 concludes. Two appendices complete the picture: Appendix A gives an alternative signed-range construction based on conjugate co- Z addition, and Appendix B collects further arithmetic optimizations whose implementation is currently underway.

2 Preliminaries

This section establishes the notation and background used throughout the paper. We define EC-ElGamal encryption to motivate the decryption problem, review elliptic curve arithmetic in affine and mixed Jacobian-affine coordinates (the latter being central to our optimizations), present the Baby-Step Giant-Step algorithm, describe Montgomery batch inversion which underlies both FastECDLP and our windowed approach, and summarize FastECDLP as the baseline for our work.

2.1 EC-ElGamal Encryption

EC-ElGamal encryption [13] is an additively homomorphic public-key scheme over an elliptic curve group. Let G be a generator of prime order q . A key pair is (sk, pk) where $sk \xleftarrow{\$} \mathbb{Z}_q$ and $pk = sk \cdot G$. To encrypt a plaintext $m \in \mathbb{Z}_q$, sample $r \xleftarrow{\$} \mathbb{Z}_q$ and compute:

$$\text{Enc}_{pk}(m) = (C_1, C_2) = (r \cdot G, m \cdot G + r \cdot pk).$$

Ciphertexts are additively homomorphic, i.e.,

$$\text{Enc}(m_1) \oplus \text{Enc}(m_2) = \text{Enc}(m_1 + m_2),$$

where the operation is performed component-wise. This property allows validators to update encrypted balances on the ledger without decrypting them. To decrypt a ciphertext, the recipient first computes $m \cdot G = C_2 - sk \cdot C_1$. Recovering m from the group element $m \cdot G$ then requires solving a small-range ECDLP, which is the problem addressed by our solver.

2.2 Elliptic Curve Arithmetic

Let E be an elliptic curve in short Weierstrass form $y^2 = x^3 + ax + b$ over a prime field $\mathbb{F}_p$, with generator G of prime order q . Our approach applies to any prime-order elliptic curve. Group elements are encoded as

33-byte compressed points. A point in affine coordinates $P \in E(\mathbb{F}_p)$ is a pair $(x, y) \in \mathbb{F}_p^2$ satisfying $y^2 = x^3 + 7 \pmod{p}$. Addition of $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ with $P_1 \neq \pm P_2$ proceeds as:

$$\lambda = (y_2 - y_1)(x_2 - x_1)^{-1}, \quad x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1,$$

incurring one field inversion. On **secp256k1**:

$$1 \text{ field inversion} \approx 3,079 \text{ ns}, \quad 1 \text{ field multiplication} \approx 26 \text{ ns}, \quad \text{ratio} \approx 119 \times^1$$

Repeated affine additions are therefore inversion-heavy.

A point in Jacobian coordinates P is represented as $(X : Y : Z)$ with $Z \neq 0$, corresponding to the affine point $(X/Z^2, Y/Z^3)$. The identity has $Z = 0$. A mixed Jacobian–affine addition of a Jacobian point $(X_1 : Y_1 : Z_1)$ and an affine point (x_2, y_2) costs $8M + 3S$ (8 multiplications, 3 squarings, no inversion) is given in Alg. 1.

Algorithm 1 Mixed Jacobian–affine addition (JacAffADD) [18]

Input: $P_1 = (X_1 : Y_1 : Z_1)$ in Jacobian, $P_2 = (x_2, y_2)$ in affine

Output: $P_3 \leftarrow \text{JacAffADD}(P_1, P_2)$ where $P_3 \leftarrow P_1 + P_2 = (X_3 : Y_3 : Z_3)$ in Jacobian

```

function JacAffADD( $P_1, P_2$ )
     $U_2 \leftarrow x_2 Z_1^2, \quad V_2 \leftarrow y_2 Z_1^3,$ 
     $H \leftarrow U_2 - X_1, \quad D \leftarrow V_2 - Y_1,$ 
     $Z_3 \leftarrow Z_1 H,$ 
     $X_3 \leftarrow D^2 - H^3 - 2X_1 H^2,$ 
     $Y_3 \leftarrow D(X_1 H^2 - X_3) - Y_1 H^3.$ 
end function
```

The affine x -coordinate is recovered as $x = X \cdot (Z^{-1})^2$, costing $1I + 1M + 1S$ (one inversion, one multiplication, and one squaring). The key property exploited in this paper is that a sequence of mixed Jacobian–affine additions costs $8M + 3S$ per step with no inversion [18, 2]. The Z -coordinates of n independently accumulated Jacobian points can all be inverted in a single batch at the cost of $1I + 3(n-1)M$ (1 inversion and $3(n-1)$ multiplications) via the Montgomery batch trick as explained in Section 2.3. In addition, consider an elliptic-curve group of prime order q defined over a field of order p . For any point $P = (x, y)$, its negation is $-P = (x, -y \pmod{p})$ share the same x -coordinate. In particular, $-(i \cdot G) = (q-i) \cdot G$, so $i \cdot G$ and $(q-i) \cdot G$ have the same x -coordinate. This symmetry is the basis for the baby-step table halving described in Section 2.5.

2.3 Montgomery Batch Inversion

Given n field elements $z_1, \dots, z_n \in \mathbb{F}_p^*$, their inverses can be computed simultaneously at the cost of a single inversion plus $3(n-1)$ multiplications, rather than n individual inversions. The algorithm, due to Montgomery [24], proceeds as follows. Compute the prefix products $p_1 = z_1, p_k = p_{k-1} \cdot z_k$ for $k = 2, \dots, n$. The final product is $P = p_n = \prod_{k=1}^n z_k$. Compute P^{-1} using one field inversion. For $k = n, n-1, \dots, 2$, compute $z_k^{-1} = p_{k-1} \cdot P^{-1}$ and update $P^{-1} \leftarrow P^{-1} \cdot z_k$. Finally, $z_1^{-1} = P^{-1}$. Total cost: $1I + 3(n-1)M$ for all n inverses.

Algorithm 2 gives the pseudocode. The two-pass structure is visible: a forward pass builds prefix products (lines 2–4), a single inversion is applied to the total product (line 5), and a backward pass extracts individual inverses (lines 6–9).

¹Minimum timings across 100,000 trials on Apple M1 Pro using **bench_field.c**, constant-time **secp256k1_fe_inv**. Minimum is used to filter OS scheduling noise on the hybrid CPU architecture (8 performance + 2 efficiency cores).

Algorithm 2 Montgomery Batch Inversion (BatchInvert) [24]

Input: $z_1, \dots, z_n \in \mathbb{F}_p^*$ **Output:** $z_1^{-1}, \dots, z_n^{-1}$

```
1:  $p_1 \leftarrow z_1$ 
2: for  $k = 2, \dots, n$  do
3:    $p_k \leftarrow p_{k-1} \cdot z_k$  ▷ forward pass: prefix products (1M)
4: end for
5:  $P^{-1} \leftarrow p_n^{-1}$  ▷ single field inversion (1I)
6: for  $k = n, n-1, \dots, 2$  do
7:    $z_k^{-1} \leftarrow p_{k-1} \cdot P^{-1}$  ▷ backward pass (2M)
8:    $P^{-1} \leftarrow P^{-1} \cdot z_k$ 
9: end for
10:  $z_1^{-1} \leftarrow P^{-1}$ 
11: return  $z_1^{-1}, \dots, z_n^{-1}$ 
```

2.4 Baby-Step Giant-Step

The baby-step giant-step (BSGS) algorithm is a generic algorithm for solving the elliptic-curve discrete logarithm problem (ECDLP) with $O(\sqrt{\ell})$ time and memory complexity, where ℓ denotes the size of the discrete logarithm search space. Given an elliptic curve group with generator G and a point $P_m = m \cdot G$ with $m \in [0, 2^\ell)$, BSGS recovers the scalar m by decomposing it as $m = jM + i$, where $\ell_1 + \ell_2 = \ell$, $M = 2^{\ell_1}$, $i \in [0, M)$, $j \in [0, 2^{\ell_2})$. Then:

$$P_m - j \cdot M \cdot G = i \cdot G,$$

yielding a meet-in-the-middle strategy. Baby steps precompute the multiples of G and store them in the table $T_1 = \{(i \cdot G, i) \mid i \in [1, M)\}$. Giant steps compute $Q_j = P_m - j \cdot M \cdot G$ and look up Q_j in T_1 , for $j = 0, 1, \dots, 2^{\ell_2} - 1$. A table hit yields $m = j \cdot M + i$ as the solution of ECDLP. Algorithm 3 gives the baseline BSGS pseudocode where $\text{SEARCH}(T'_1, x)$ denotes a cuckoo hash lookup of x in the baby table T'_1 , returning the stored index i and a boolean ok indicating whether a match was found. This notation is used throughout the paper.

Algorithm 3 Baby-Step Giant-Step (baseline)

Input: $\ell_1, \ell_2, \ell = \ell_1 + \ell_2$, $T_1 = \{(i \cdot G, i) \mid i \in [0, 2^{\ell_1})\}$, $P_m = m \cdot G$ with $m \in [0, 2^\ell)$, $M = 2^{\ell_1}$ **Output:** m or $\perp$

```
1:  $(Q, step) \leftarrow (P_m, M \cdot G)$ 
2: for  $j = 0, 1, \dots, 2^{\ell_2} - 1$  do
3:    $(i, ok) \leftarrow \text{SEARCH}(T'_1, Q)$ 
4:   if  $ok$  then
5:     return  $j \cdot M + i$ 
6:   end if
7:    $Q \leftarrow Q - step$  ▷ affine subtraction: 1 inversion
8: end for
9: return  $\perp$ 
```

2.5 FastECDLP

FastECDLP [30] is the current state-of-the-art algorithm for solving small ECDLP instances, combining BSGS with three key optimizations.

Baby-step table compression. The baby-step table T'_1 stores only x -coordinates by exploiting the fact that $i \cdot G$ and $-(i \cdot G)$ share the same x -coordinate. A table hit with stored index i yields two candidates, $m = j \cdot M + i$ and $m = j \cdot M - i$. This allows the baby-step table to store entries only for $i \in [1, M/2]$ instead

of all $i \in [1, M)$. Indeed, any remainder $r \in (M/2, M)$ satisfies $m = (j + 1) \cdot M - (M - r)$ and is found at step $j + 1$ via the $-i$ candidate with $i = M - r \in (0, M/2)$. Furthermore, the baby-step table stores only the upper 64 bits of the affine x -coordinate as the lookup key, rather than the full 33-byte compressed point. Combined with the negation map, this reduces the baby-step table from $M \times 33$ bytes to $(M/2) \times 8$ bytes, an $\approx 8\times$ memory reduction. Furthermore, the baby-step table stores only 64 bits for the hash values and the keys to ensure finding the correct i from T'_1 , rather than the full 33-byte compressed point. Combined with the negation map, this reduces the baby-step table from $M \times 33$ bytes to $(M/2) \times 8$ bytes, an $\approx 8\times$ memory reduction. The baby-step table uses $k = 3$ cuckoo hashing with a load factor of $1/1.3$, while providing worst-case $O(1)$ lookup with exactly three probes.

Giant-step batch inversion. The full giant-step table $T'_2 = \{j \cdot M \cdot G \mid j \in [1, 2^{\ell_2-1}]\}$ is precomputed in affine coordinates. A table hit yields four candidates $m = \pm j \cdot M \pm i$, all verified against P_m at negligible cost. Precomputing T'_2 makes all denominators $z_j = T'_2[j][x] - P_m[x]$ available simultaneously. Montgomery batch inversion (TreeMon) is then applied to all 2^{ℓ_2-1} values at a cost of one inversion and $3(2^{\ell_2-1} - 1)$ multiplications, giving amortised cost ≈ 3 multiplications per element, a $\approx 34\times$ reduction over individual inversions at $\ell_2 = 24$. On an Intel Xeon Gold 5218 processor running at 2.30 GHz with 16 threads, FastECDLP [30] reports execution times of 248 ms for a 52-bit search space and 990 ms for a 54-bit search space.

Unscalable memory requirements. Storing T'_2 requires 2^{ℓ_2-1} affine points, with each point occupying 80 bytes in the Go implementation of Tang et al. [30] (two `FieldVal` field elements of 40 bytes each). For a 64-bit search space ($\ell_1 = 31$, $\ell_2 = 33$), this requires approximately 344 GB of memory.² Similarly, a C implementation using the `secp256k1` library, which we present in the following sections, requires 88 bytes per affine point (two `secp256k1_fe` field elements of 40 bytes each plus an infinity indicator), resulting in approximately 378 GB of memory for the same search space. These memory requirements far exceed the capacity of a typical workstation. Thus, as noted by Tang et al., searching a 64-bit range remains impractical on a typical workstation due to the memory requirements of the giant-step table T'_2 , a limitation we address in this paper.

Algorithm 4 gives the pseudocode of the original FastECDLP. In the `SEARCH` calls, the lookup key is a 32-bit portion of the affine x -coordinate $P[x]$, rather than the full 33-byte point. The details can be found in Section 5.1. Note that lines 6, 14, 15, and 20 of Alg. 4 read directly from the giant-step table T'_2 . Therefore, T'_2 must be fully precomputed in affine coordinates before line 6, and no Jacobian arithmetic is performed during a solve. These two observations motivate our contributions in Section 3.

2.6 Unsigned-to-Signed Range Conversion

FastECDLP [30] operates on a signed plaintext range $m' \in [-2^{\ell-1}, 2^{\ell-1})$, which halves the giant-step table T'_2 by covering both positive and negative candidates with a single pass. In practice, AHE applications such as confidential token transfers use unsigned plaintexts $m \in [0, 2^\ell)$. We apply a one-time conversion described in Algorithm 5 before invoking any signed solver, making the signed formulation transparent to the application layer.

Algorithm 5 Unsigned-to-Signed Wrapper

Input: P_m , ℓ , `SIGNEDSOLVER` returning $m' \in [-2^{\ell-1}, 2^{\ell-1})$ for $P_{m'} = m' \cdot G$

Output: $m \in [0, 2^\ell)$ such that $P_m = m \cdot G$

- 1: $offset \leftarrow 2^{\ell-1}$
 - 2: $P_{m'} \leftarrow P_m - offset \cdot G$ $\triangleright$ one EC subtraction, negligible cost
 - 3: $m' \leftarrow \text{SIGNEDSOLVER}(P_{m'}, \ell)$
 - 4: **return** $(m' + offset)$
-

²Although the authors state that in general T'_2 would require $(2^{\ell_2-1} \cdot 33)$ bytes of memory, the Go implementation accompanying the paper stores both the x - and y -coordinates in `T2x` and `T2y`.

Algorithm 4 FastECDLP [30] (signed range)

Input: $\ell_1, \ell_2, \ell = \ell_1 + \ell_2$, $T'_1 = \{\langle P[x], i \rangle \mid P = i \cdot G, i \in [1, 2^{\ell_1-1}]\}$ with $k=3$ cuckoo hashing and 8-byte packed entries (lookup key = upper 64 bits of x -coordinate), $T'_2 = \{j \cdot M \cdot G \mid j \in [1, 2^{\ell_2-1}]\}$ precomputed in affine (where $M = 2^{\ell_1}$), $P_m = m \cdot G$ with $m \in [-2^{\ell-1}, 2^{\ell-1}]$

Output: m or $\perp$

```
1:  $(i, ok) \leftarrow \text{SEARCH}(T'_1, P_m[x])$ 
2: if  $ok$  then
3:   return  $m \in \{i, -i\}$  s.t.  $m \cdot G = P_m$ 
4: end if
5: for  $j = 1, 2, \dots, 2^{\ell_2-1}$  do ▷ collect denominators
6:    $z_j \leftarrow T'_2[j][x] - P_m[x] \bmod p$  ▷  $T'_2$  fully precomputed before this line
7:   if  $z_j = 0$  then ▷  $P_m = j \cdot M \cdot G$ 
8:     return  $m \in \{j \cdot M, -(j \cdot M)\}$  s.t.  $m \cdot G = P_m$ 
9:   end if
10:   $Z \leftarrow Z \cup \{z_j\}$ 
11: end for
12:  $Z' \leftarrow \text{TreeMon}(Z)$  ▷ 1 inv +  $3(2^{\ell_2-1}-1)$  mults
13: for  $j = 1, 2, \dots, 2^{\ell_2-1}$  do
14:   $\varphi \leftarrow P_m[x] + T'_2[j][x] \bmod p$ 
15:   $Q[x] \leftarrow ((p - T'_2[j][y] - P_m[y]) \cdot z'_j)^2 - \varphi \bmod p$  ▷  $Q = P_m - T'_2[j]$ 
16:   $(i, ok) \leftarrow \text{SEARCH}(T'_1, Q[x])$ 
17:  if  $ok$  then
18:    return  $m \in \{j \cdot M + i, j \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
19:  end if
20:   $Q[x] \leftarrow ((T'_2[j][y] - P_m[y]) \cdot z'_j)^2 - \varphi \bmod p$  ▷  $Q = P_m + T'_2[j]$ 
21:   $(i, ok) \leftarrow \text{SEARCH}(T'_1, Q[x])$ 
22:  if  $ok$  then
23:    return  $m \in \{-j \cdot M + i, -j \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
24:  end if
25: end for
26: return  $\perp$ 
```

The conversion costs one EC point subtraction and one integer addition, both negligible compared to the solver cost. All signed implementations in this paper as described in Section 5 (`fastecdip_treemon.c`, `fastecdip_parallel.c`, `bsgs_dlp_benchmark_cached_zaddsub.c`, `bsgs_dlp_benchmark_cached_zaddc.c`) use this wrapper to accept unsigned inputs while operating internally on the signed range.

3 Jacobian Coordinate Loop and Windowed-Batch Inversion

We present two algorithmic contributions that together eliminate the precomputed giant-step table and enable full-range decryption. The first keeps the giant-step running point in Jacobian coordinates, removing the need to precompute the giant-steps in affine coordinates while reducing per-step inversions. The second batch-inverts a local window of W accumulated Z -coordinates at once, reducing memory from $O(2^{\ell_2})$ to $O(W)$ and making 64-bit decryption feasible.

3.1 Jacobian Coordinate Loop

FastECDLP precomputes the giant steps $j \cdot M \cdot G$ in affine coordinates while avoiding an inversion at each step by using Montgomery batch inversion. As the memory required to store the giant-step table T'_2 increases with ℓ_2 , storing the table becomes impractical for larger values of ℓ . To reduce the memory requirement, one may instead compute the giant steps $j \cdot M \cdot G$ on the fly. Advancing these points in affine coordinates requires one inversion per step, which must be resolved immediately and therefore cannot be amortised across

multiple steps. We instead advance the giant steps in Jacobian coordinates using mixed Jacobian–affine point addition (zero inversions per step). This defers all inversions to the end of a window of W steps, where they can be computed together via a single batch inversion as described in Section 3.2.

Algorithm 6 gives the pseudocode. Compared to Algorithm 3, the key change is that Q is advanced using JACADD (line 9) with no inversion, and the single inversion per step is deferred to TOAFFINE at line 4 only for the lookup. There is no T'_2 table.

Algorithm 6 BSGS with Jacobian Coordinate Loop (our §3.1)

Input: $\ell_1, \ell_2, T'_1 = \{\langle i \cdot G[x], i \rangle \mid i \in [1, 2^{\ell_1-1}]\}$, $P_m = m \cdot G$, $M = 2^{\ell_1}$, $S = M \cdot G$ in affine

Output: m or $\perp$

```

1:  $Q \leftarrow P_m$  ▷  $Q$  kept in Jacobian throughout
2:  $negS \leftarrow -S$  ▷ affine negation, one-time
3: for  $j = 0, 1, \dots, 2^{\ell_2}$  do
4:    $Q_{\text{aff}} \leftarrow \text{TOAFFINE}(Q)$  ▷ 1 inversion for lookup only
5:    $(i, ok) \leftarrow \text{SEARCH}(T'_1, Q_{\text{aff}}[x])$ 
6:   if  $ok$  then
7:     return  $m \in \{j \cdot M + i, j \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
8:   end if
9:    $Q \leftarrow \text{JacAffADD}(Q, negS)$  ▷ mixed Jacobian–affine, 0 inversions
10: end for
11: return  $\perp$ 

```

3.2 Windowed-Batch Inversion

The Jacobian loop requires one inversion per step. We improve this following a windowed batch inversion technique: instead of calling the SEARCH procedure in each iteration, which requires an inversion for affine conversion, we accumulate the Z -coordinates for every W iterations. Then, after every W iterations we batch-invert these accumulated Z -coordinates in one shot with a cost of 1 inversion + $3(W-1)$ multiplications for all W lookups reducing the per-step inversion cost to $1/W$. Crucially, this approach does not require precomputing T'_2 . The giant-step point $Q_j = P_m - j \cdot M \cdot G$ is advanced iteratively in Jacobian coordinates, accumulating W points per window. The algorithm is given in Algorithm 7. Note that the cost is

$$C(W) \approx \underbrace{297 \text{ ns}}_{\text{Jac. add}} + \underbrace{\frac{3,094}{W} \text{ ns}}_{\text{inv. share}} + \underbrace{90 \text{ ns}}_{3 \text{ mults/step}}.$$

The inversion term is negligible for $W \gtrsim 8$. The optimal W is bounded by the L2 cache capacity of the $W \times 96$ -byte per-thread workspace:

bits	ℓ_2	Optimal W
52	22	256
54	23	512
58	27	512

Algorithm 7 and FastECDLP lie on the same continuum parameterised by window size W . FastECDLP is the $W = 2^{\ell_2-1}$ endpoint: it precomputes all Z -coordinates in advance by storing T'_2 , enabling a single batch inversion over the entire range. Our approach occupies the memory-constrained regime $W \ll 2^{\ell_2}$, eliminating T'_2 at the cost of $\lceil 2^{\ell_2}/W \rceil$ inversions instead of one. At $W = 512$ this cost is negligible relative to the Jacobian addition cost.

Algorithm 7 Windowed-Batch Inversion BSGS for $m \in [0, 2^\ell)$

Input: $P_m = (x_m : y_m : 1)$, baby-step table T'_1 , step $S = M \cdot G = (x_S, y_S)$, window W **Output:** m or $\perp$

```
1:  $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_m)$ 
2: if  $ok$  then
3:   return  $m \leftarrow i$ 
4: end if
5:  $Q \leftarrow P_m$ ;  $negS \leftarrow -S$ ;  $buf \leftarrow []$ 
6: for  $j = 1$  to  $2^{\ell_2}$  do
7:    $Q \leftarrow \text{JacAffADD}(Q, negS)$   $\triangleright 8M + 3S$ 
8:   if  $Q = \mathcal{O}$  then  $\triangleright P_m = j \cdot M \cdot G$ 
9:     return  $m \leftarrow j \cdot M$ 
10:  end if
11:   $buf.append((Q, j))$ 
12:  if  $|buf| = W$  or  $j = 2^{\ell_2}$  then
13:     $\{Z_k^{-1}\} \leftarrow \text{BatchInvert}(\{Z_k\})$   $\triangleright 1I + 3(W-1)M$ 
14:    for each  $(Q_k = (X_k : Y_k : Z_k), j_k)$  in  $buf$  do
15:       $x_k \leftarrow X_k \cdot (Z_k^{-1})^2 \bmod p$   $\triangleright 1M + 1S$ 
16:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
17:      if  $ok$  then
18:        return  $m \in \{j_k \cdot M + i, j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
19:      end if
20:    end for
21:     $buf \leftarrow []$ 
22:  end if
23: end for
```

Cost Analysis. The cost of the worst-case where BSGS requires the maximum number of iterations (2^{ℓ_2}) is as follows:

$$\begin{aligned} Cost &= 2^{\ell_2} \times \left((8M + 3S) + \frac{1}{W}(1I + 3(W-1)M) + (1M + 1S) + 1L \right) \\ &\approx 2^{\ell_2} \times \left((12M + 4S) + \frac{1I}{W} + 1L \right) \\ &\approx \frac{2^{\ell_2}}{2} \times \left((24M + 8S) + 2 \cdot \frac{1I}{W} + 2L \right) \end{aligned}$$

where L denotes the cost of a look-up operation and the last line is rewritten to facilitate comparison with the signed Co-Z solver below.

3.3 Windowed-Batch Inversion BSGS Adapting to Signed Range $[-2^{\ell-1}, 2^{\ell-1})$

In this section, we explore adapting the windowed-batch inversion BSGS, proposed in the previous section, to work with the signed range in order to enhance its performance. To this end, we propose two distinct algorithmic structures: one is presented in this section, and the other is detailed in Appendix A.

Recall that the primary benefit of operating within the signed range, as shown in Section 2.5, is that it allows two simultaneous giant steps per iteration: one step in the backward direction ($Q \leftarrow P_m - T'_2[j]$ as in line 15) and another step in the forward direction ($Q \leftarrow P_m + T'_2[j]$ as in line 20). This allows us to reuse common operations, such as performing a single modular inversion instead of two, and reduces the size of the precomputed giant-steps table.

In our windowed-batch inversion BSGS for the unsigned range, we only track the backward direction ($Q \leftarrow Q - S$ as in line 7). Applying both directions requires maintaining two separate EC points: Q_{sub}

for the backward step and Q_{add} for the forward step. At the beginning, both points are initialized to P_m , similar to line 5. In each iteration, $Q_{sub} \leftarrow Q_{sub} - S$ via $\text{JacAffSUB}(Q_{sub}, S)$, and $Q_{add} \leftarrow Q_{add} + S$ via $\text{JacAffADD}(Q_{add}, S)$. However, directly applying these giant steps yields two independent EC points. During the baby-step lookup, these points demand separate Z -inversions (costing $\approx 3M$ each), which reduces overall performance.

Combined mixed Jacobian–affine Addition and Subtraction. To mitigate this overhead and leverage the common giant step shared across directions, we optimize at the arithmetic layer. Given two Jacobian EC points P_1 and P_2 sharing a Z -coordinate, along with an affine point P , the goal is to simultaneously compute $P_a \leftarrow P_1 + P$ and $P_s \leftarrow P_2 - P$ such that P_a and P_s maintain a common Z -coordinate. This enables them to be used directly in the next iteration. Algorithm 8 states both operations side by side, exposing the intermediate terms they have in common.

Algorithm 8 Mixed Jacobian–affine addition and subtraction, side by side (the addition restates Algorithm 1).

Input: $P_1 = (X_1 : Y_1 : Z)$, $P_2 = (X_2 : Y_2 : Z)$, and $P = (x, y)$

Output: $P_a \leftarrow P_1 + P = (X_a : Y_a : Z_a)$ and $P_s \leftarrow P_2 - P = (X_s : Y_s : Z_s)$

```

function JacAffADD( $P_1, P$ )
   $U_a \leftarrow xZ^2$ ,       $V_a \leftarrow yZ^3$ ,
   $H_a \leftarrow U_a - X_1$ ,  $D_a \leftarrow V_a - Y_1$ ,
   $Z_a \leftarrow ZH_a$ ,
   $X_a \leftarrow D_a^2 - H_a^3 - 2X_1H_a^2$ ,
   $Y_a \leftarrow D_a(X_1H_a^2 - X_a) - Y_1H_a^3$ .
end function

```

```

function JacAffSUB( $P_2, P$ )
   $U_s \leftarrow xZ^2$ ,       $V_s \leftarrow -yZ^3$ ,
   $H_s \leftarrow U_s - X_2$ ,  $D_s \leftarrow V_s - Y_2$ ,
   $Z_s \leftarrow ZH_s$ ,
   $X_s \leftarrow D_s^2 - H_s^3 - 2X_2H_s^2$ ,
   $Y_s \leftarrow D_s(X_2H_s^2 - X_s) - Y_2H_s^3$ .
end function

```

Algorithm 9 Mixed Jacobian–affine addition and subtraction (JacAffADDSUB)³

Input: $P_1 = (X_1 : Y_1 : Z)$, $P_2 = (X_2 : Y_2 : Z)$ and $P = (x, y)$

Output: $(P_a, P_s) \leftarrow \text{JacAffADDSUB}(P_1, P_2, P)$ where $P_a \leftarrow P_1 + P = (\tilde{X}_a : \tilde{Y}_a : Z_c)$ and $P_s \leftarrow P_2 - P = (\tilde{X}_s : \tilde{Y}_s : Z_c)$

```

function JacAffADDSUB( $P_1, P_2, P$ )
   $U \leftarrow xZ^2$ ,       $V \leftarrow yZ^3$ ,
   $H_a \leftarrow U - X_1$ ,  $H_s \leftarrow U - X_2$ ,
   $D_a \leftarrow V - Y_1$ ,  $D_s \leftarrow -V - Y_2$ ,
   $X_a \leftarrow D_a^2 - H_a^3 - 2X_1H_a^2$ ,
   $X_s \leftarrow D_s^2 - H_s^3 - 2X_2H_s^2$ ,
   $Y_a \leftarrow D_a(X_1H_a^2 - X_a) - Y_1H_a^3$ ,
   $Y_s \leftarrow D_s(X_2H_s^2 - X_s) - Y_2H_s^3$ ,
   $Z_c \leftarrow ZH_aH_s$ ,
   $\tilde{X}_a \leftarrow H_s^2X_a$ ,       $\tilde{X}_s \leftarrow H_a^2X_s$ ,
   $\tilde{Y}_a \leftarrow H_s^3Y_a$ ,       $\tilde{Y}_s \leftarrow H_a^3Y_s$ 
end function

```

$Cost = 2(8M + 3S) - (3M + 1S) + (4M)$
 $= 17M + 5S$

```

function JacAffADDSUB( $P_1, P_2, P$ )
   $U \leftarrow xZ^2$ ,       $V \leftarrow yZ^3$ 
   $H_a \leftarrow U - X_1$ ,  $H_s \leftarrow U - X_2$ 
   $D_a \leftarrow V - Y_1$ ,  $D_s \leftarrow -V - Y_2$ 
   $H_c \leftarrow H_aH_s$ ,       $Z_c \leftarrow ZH_c$ 
   $W_U \leftarrow UH_c^2$ 
   $W_1 \leftarrow X_1H_c^2$ ,       $W_2 \leftarrow X_2H_c^2$ 
   $K_1 \leftarrow Y_1H_c^3$ ,       $K_2 \leftarrow Y_2H_c^3$ 
   $L_a \leftarrow D_aH_s$ ,       $L_s \leftarrow D_sH_a$ 
   $\tilde{X}_a \leftarrow L_a^2 - W_U - W_1$ ,  $\tilde{X}_s \leftarrow L_s^2 - W_U - W_2$ 
   $\tilde{Y}_a \leftarrow L_a(W_1 - \tilde{X}_a) - K_1$ ,  $\tilde{Y}_s \leftarrow L_s(W_2 - \tilde{X}_s) - K_2$ 
end function

```

$Cost^4 = 15M + 4S$

Because the input Z -coordinate and the affine point are identical for both operations, reusing the intermediate terms $U \leftarrow xZ^2$ and $V \leftarrow yZ^3$ reduces the cost by $3M + 1S$. Furthermore, P_a and P_s can share a common Z -coordinate ($Z_c \leftarrow ZH_aH_s$), but this requires internal scaling of the X - and Y -coordinates by factors H_s^2

³Special cases such as $P_1 = \pm P$, $P_2 = \pm P$, or $P_1, P_2 = \mathcal{O}$ are handled in the implementation.

⁴The cost can be reduced by one further multiplication to $14M + 4S$; the implementation and timings reported in Section 5 correspond to the $15M + 4S$ variant given here.

and H_s^3 for P_a , and by H_a^2 and H_a^3 for P_s . Since these factors are already computed, the rescaling costs only 4M. The left side of Algorithm 9 demonstrates combining both addition and subtraction into a single function (JacAffADDSUB), as described above. The cost of this new function is 17M + 5S. The right side of Algorithm 9 presents a fully optimized version that reduces the total cost to 15M + 4S, which is 1M + 2S less than performing two independent operations.

BSGS for signed range using Combined mixed Jacobian–affine Addition and Subtraction.

Algorithm 10 adapts Algorithm 7 to work with the signed range by utilizing the JacAffADDSUB function. As a result, the maximum number of required giant steps is reduced from 2^{ℓ_2} to 2^{ℓ_2-1} . Because Q_{sub} and Q_{add} share a common Z -coordinate, each iteration requires only one Z -inversion. This cuts operational costs by $\approx 3M$ for the inversion itself and 1S for computing $(Z^{-1})^2$.

Algorithm 10 Windowed-Batch Inversion BSGS for $m \in [-2^{\ell_2-1}, 2^{\ell_2-1})$

Input: $P_m = (x_m : y_m : 1)$, baby-step table T'_1 , step $S = M \cdot G = (x_S, y_S)$, window W

Output: m or $\perp$

```

1:  $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_m)$ 
2: if  $ok$  then
3:   return  $m \in \{i, -i\}$  s.t.  $m \cdot G = P_m$ 
4: end if
5:  $Q_{sub} \leftarrow P_m, Q_{add} \leftarrow P_m; \text{ buf} \leftarrow []$ 
6: for  $j = 1$  to  $2^{\ell_2-1}$  do
7:    $(Q_{add}, Q_{sub}) \leftarrow \text{JacAffADDSUB}(Q_{add}, Q_{sub}, S)$  ▷ 15M + 4S
8:   if  $Q_{sub} = \mathcal{O}$  then
9:     return  $m \leftarrow j \cdot M$ 
10:  end if
11:  if  $Q_{add} = \mathcal{O}$  then
12:    return  $m \leftarrow -j \cdot M$ 
13:  end if
14:   $\text{buf.append}(((X_{add}, X_{sub}, Z), j))$ 
15:  if  $|\text{buf}| = W$  or  $j = 2^{\ell_2-1}$  then
16:     $\{Z_k^{-1}\} \leftarrow \text{BatchInvert}(\{Z_k\})$  ▷ 1I + 3(W-1)M
17:    for each  $((X_{add_k}, X_{sub_k}, Z_k), j_k)$  in  $\text{buf}$  do
18:       $x_k \leftarrow X_{sub_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M + 1S
19:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
20:      if  $ok$  then
21:        return  $m \in \{j_k \cdot M + i, j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
22:      end if
23:       $x_k \leftarrow X_{add_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M
24:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
25:      if  $ok$  then
26:        return  $m \in \{-j_k \cdot M + i, -j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
27:      end if
28:    end for
29:     $\text{buf} \leftarrow []$ 
30:  end if
31: end for

```

Cost Analysis. The cost of the worst-case where BSGS requires the maximum number of iterations (2^{ℓ_2-1}) is as follows:

$$\begin{aligned} Cost &= \frac{2^{\ell_2}}{2} \times \left((15M + 4S) + \frac{1}{W}(11 + 3(W-1)M) + (2M + 1S) + 2L \right) \\ &\approx \frac{2^{\ell_2}}{2} \times \left((20M + 5S) + \frac{11}{W} + 2L \right) \end{aligned}$$

Therefore, the signed range algorithm requires $\frac{2^{\ell_2}}{2} \times \left((4M + 3S) + \frac{11}{W} \right)$ operations less than the unsigned range algorithm for the worst-case, which is equivalent to up to 19% improvement.

4 Discussion

The cost analysis so far treats each decryption as an isolated instance in which the plaintext may lie anywhere in $[0, 2^\ell)$ and nothing further is known about it. Many applications offer more. Plaintexts frequently arrive in groups rather than one at a time. An auditor opening every balance on a confidential-transfer ledger, for instance, faces a set of values that is constrained in total even though no individual value is. Side information about a single plaintext is equally common: past behaviour may indicate the range in which a value is likely to lie, and the encoding itself may force values to be multiples of a known factor. This section examines what such structure buys. We first bound the total work of recovering n amounts whose sum is itself bounded by the search range, showing that the entire batch costs little more than a single worst-case solve. We then turn to single instances whose plaintext range or arithmetic structure is partly known, where the giant-step search can be reordered or shortened without any change to the baby-step table or to the memory footprint of the solver.

4.1 Solver Time for Multiple Amounts

An issuer or auditor may need to recover n amounts simultaneously after EC ElGamal decryption. We present a theoretical bound on the total solver work and a practical batch algorithm.

Proposition 1 (Batch Solver Time Bound). *Let $m_1, m_2, \dots, m_n$ be n amounts with $m_i \in [0, 2^\ell)$ for all i , satisfying the sum constraint $\sum_{i=1}^n m_i \leq 2^\ell - 1$. With BSGS parameters $M = 2^{\ell_1}$ and $\ell_2 = \ell - \ell_1$, the total number of giant steps across all n independent solves using Algorithm 7 is strictly bounded by:*

$$\sum_{i=1}^n \left\lceil \frac{m_i}{M} - \frac{1}{2} \right\rceil < 2^{\ell_2} + n.$$

In particular, for $n \ll 2^{\ell_2}$, the total solver work across all n instances is asymptotically bounded by the cost of a single worst-case solve, regardless of how the total value is distributed among the n amounts.

Proof. For each amount m_i , the number of giant steps satisfies

$$\left\lceil \frac{m_i}{M} - \frac{1}{2} \right\rceil \leq \left\lceil \frac{m_i}{M} \right\rceil \leq \frac{m_i}{M} + 1.$$

Summing over all n instances gives

$$\sum_{i=1}^n \left\lceil \frac{m_i}{M} - \frac{1}{2} \right\rceil \leq \left(\frac{1}{M} \sum_{i=1}^n m_i \right) + n.$$

Applying the sum constraint $\sum_{i=1}^n m_i \leq 2^\ell - 1$ and the parameter relation $M \cdot 2^{\ell_2} = 2^\ell$, we obtain

$$\frac{2^\ell - 1}{M} + n = 2^{\ell_2} - \frac{1}{M} + n < 2^{\ell_2} + n,$$

which yields the final bound. □

Remark 1. In the XRPL MPT setting with $\ell = 63$, the sum of the corresponding balances satisfies $\sum_{i=1}^n m_i \leq 2^{63} - 1$. Proposition 1 guarantees that the total solver work for an auditor or issuer recovering n amounts is bounded by a single worst-case 63-bit solve, regardless of how the total balance is distributed among the n accounts.

Parallel batch execution. While the batch solver bound of Proposition 1 directly dictates the total sequential cost, the batch format allows the solver to naturally exploit $T > 1$ threads by assigning one independent ECDLP instance to each thread, rather than dividing the giant-step range of a single instance across threads. We assign one independent ECDLP instance to each thread. The baby-step table T'_1 is shared read-only among all threads, while each worker maintains its own Jacobian running point and $O(W)$ window workspace. When a thread completes an instance, it immediately pulls the next unsolved instance from a shared work queue. This strategy avoids redundant work that occurs when multiple threads search different portions of the same giant-step range (intra-instance parallelism). Because each thread performs only the necessary walks for its specific instance, this approach perfectly preserves the aggregate-work property of the batch. Consequently, for a large batch ($n \gg T$), the workload distributes efficiently across the threads. The parallel execution time scales down linearly, meaning the entire batch can be solved in roughly the same time it would take to solve a single worst-case instance using T threads. To prevent thread starvation near the end of the execution, when fewer than T unsolved instances remain in the queue, the solver can dynamically switch back to intra-instance parallelism. Idle threads are reassigned to search disjoint giant-step ranges of the final unresolved instances, ensuring maximum resource utilization until the entire batch is complete.

4.2 Range-Hint and Plaintext Structure Guided Search

Range-Hint Guided Search. If the application has prior knowledge of the approximate plaintext range, the giant step search can be reorganized to prioritize the most likely range first, achieving early termination much faster than worst-case. For example, if balances are expected in the range $[2^{40}, 2^{50})$ with $\ell_1 = 31$, the corresponding giant steps are:

$$j \in \left[\frac{2^{40}}{2^{31}}, \frac{2^{50}}{2^{31}} \right) = [2^9, 2^{19})$$

Only $2^{19} - 2^9 \approx 2^{19}$ steps cover the likely range, compared to the full 2^{33} steps needed for 64-bit, a reduction of over 8,000 $\times$.

In the single-thread case, the solver searches the hint range $[j_{\min}, j_{\max})$ first. If no solution is found, it extends to the remaining range $[0, j_{\min}) \cup [j_{\max}, 2^{\ell_2})$.

For $T > 1$ threads, the hint range is divided equally with thread t searching:

$$\left[j_{\min} + t \cdot \frac{j_{\max} - j_{\min}}{T}, j_{\min} + (t+1) \cdot \frac{j_{\max} - j_{\min}}{T} \right) \quad t = 0, 1, \dots, T-1.$$

If no solution is found in Phase 1, the search extends to the remaining range $[0, j_{\min}) \cup [j_{\max}, 2^{\ell_2})$ in Phase 2, again divided equally across threads. No change to the baby table or memory footprint is required. In the XRPL context, transaction history can inform the expected balance range for a given account, making this optimization practically useful for client-side decryption.

Exploiting Known Plaintext Structure. In applications where the plaintext scalar m is known to be divisible by a factor s , the search range can be reduced by a factor of s at the cost of one scalar multiplication. Specifically, compute $P_{m'} = s^{-1} \cdot P_m$ where $s^{-1} \bmod q$ is the modular inverse of s , then solve for $m' = m/s \in [0, 2^{\ell}/s)$, reducing giant steps from 2^{ℓ_2} to $2^{\ell_2}/s$.

5 Experimental Results

All experiments run on two machines using the same `libsecp256k1` library: Apple M1 Pro (32 GB RAM, 10 cores) and an Intel Core Ultra 9 285K (64 GB RAM, 24 cores, Turbo Boost disabled). To enable a fair, same-hardware comparison we implemented six solvers in C, all sharing the same `secp256k1` library and baby table

cache files. A reproducible benchmark script (`benchmark.sh`) is included in the repository. The implementation and benchmark script are publicly available at <https://github.com/mrtcnk/Small-DLP-Solver-BSGS>.

`fastecdlp_treemon.c` is a faithful re-implementation of Tang et al. [30], matching the Go/C++ reference architecture: Phase 1 (denominator computation) runs sequentially, matching `BuildZAndTryZeroDiff` in the Tang et al. C++ source; Phase 2 uses a parallel binary product tree matching `BuildInvTree` with `yacl::parallel_for` per depth level; Phase 3 (search) is parallelised across T threads. Both `fastecdlp_treemon.c` and `fastecdlp_parallel.c` use the signed-range conversion (Section 2), halving T'_2 to 2^{ℓ_2-1} points, and include fixes for two edge cases: (i) zero denominator when $P_m = j \cdot M \cdot G$ (would cause division by zero in batch inversion), and (ii) correct baby table lookup when $T'_2[j]$ is the point at infinity.

`fastecdlp_parallel.c` applies our parallelisation contribution: Phase 1 and Phase 2 are split across T threads, each owning $2^{\ell_2-1}/T$ elements with its own batch inversion. This eliminates the sequential bottleneck at the cost of T inversions instead of one, but reduces synchronisation to a single barrier and requires no global arrays.

`fastecdlp_jacobian.c` additionally applies our Jacobian loop (Section 3.1), eliminating T'_2 entirely. Each solve performs a Jacobian walk over $[0, 2^{\ell_2})$ followed by per-thread batch inversion, with working memory $O(2^{\ell_2}/T)$ per thread.

`bsgs_dlp_benchmark_cached.c` adds windowed batch inversion (Section 3.2), reducing per-thread working memory to $O(W) \approx 50$ KB regardless of bit size.

`bsgs_dlp_benchmark_cached_zaddsub.c` applies Co-Z signed arithmetic simultaneously computing both $Q_{add} = P_m + j \cdot M \cdot G$ and $Q_{sub} = P_m - j \cdot M \cdot G$ with a shared Z -coordinate at cost $15M + 4S$ per step (Section 3.3), our fastest complete solver.

`bsgs_dlp_benchmark_cached_zaddc.c` is a variant using the `zaddc` Co-Z formula ($6M + 3S$) for comparison, which computes a different pair of outputs and requires additional operations to recover both Q_{add} and Q_{sub} (Section A).

All six implementations adopt Tang et al.’s window-based cuckoo hashing (Section IV.C.2 of [30]), as described in Section 5.1. Search times measure per-solve cost only. The one-time baby table build and the one-time T'_2 precomputation are excluded as both are independent of P_m and amortised across solves. Two benchmark modes are reported: random trials, averaged across multiple seeds, and the worst case $m = 2^\ell - 1$, the plaintext requiring the maximum number of giant steps. Trial counts are given in each subsection.

Parallelisation of the Giant-Step Search. Our Jacobian-based solvers are parallelised by dividing the giant-step range $[0, 2^{\ell_2})$ equally among T threads, each maintaining its own Jacobian running point and $O(W)$ window workspace. A shared termination flag enables early exit as soon as any thread finds the solution.

5.1 Baby Table: Packed Entries and Cuckoo Hashing

We re-implement the memory optimizations of FastECDLP [30] within our framework. The baby table uses $k = 3$ cuckoo hashing at load factor $1/1.3\times$, divided into three sections with one probe position per section; lookup checks exactly 3 positions in $O(1)$ worst-case. The factor 1.3 is the cuckoo overhead: to store $M/2$ entries, the table allocates $1.3 \times M/2$ slots across its three sections, so total baby table memory is $1.3 \times (M/2) \times 8$ bytes $= 1.3 \times M \times 4$ bytes. We use $\alpha = 1.3$ to denote this overhead factor throughout. Together with the negation map halving, these reduce the baby table from 8.59 GB (open-addressing, $\ell_1 = 30$) to 5.58 GB (cuckoo, $\ell_1 = 30$), a $1.54\times$ saving that enables $\ell_1 = 30$ on a 32 GB machine.

Following Tang et al. (Section IV.C.2 of [30]), the cuckoo hash uses the 32-byte affine x -coordinate directly as the hash input, split into three independent 8-byte windows, one per section:

Window	Bytes	Role
h_1	$X[0:4] \rightarrow \text{position}, X[4:8] \rightarrow \text{key}$	Section 0
h_2	$X[8:12] \rightarrow \text{position}, X[12:16] \rightarrow \text{key}$	Section 1
h_3	$X[16:20] \rightarrow \text{position}, X[20:24] \rightarrow \text{key}$	Section 2

For each window, the bucket position is `BytesToUint32(first 4 bytes) % section.size` and the 32-bit fingerprint key is `BytesToUint32(last 4 bytes)`. Bytes $X[24:31]$ are unused. Each section has a genuinely independent fingerprint key. Each baby table entry stores the per-section fingerprint key and the index i as a 32-bit value, giving a packed 8-byte entry. The value zero serves as the empty sentinel since $i \geq 1$.

Since each section uses only 4 bytes as the fingerprint key, all 3 cuckoo positions must be checked on every lookup to avoid false negatives caused by distinct x -coordinates sharing the same 4-byte window key. This is particularly important at 64-bit where $\sim 8 \times 10^9$ lookups are performed.

All implementations use the window-based hash throughout, following Tang et al., for faithfulness to the reference implementation.

5.2 Window Size Selection

The window size W controls the trade-off between batch inversion amortisation and per-thread workspace size. A larger W reduces the amortised inversion cost per step but increases the per-thread workspace ($W \times 96$ bytes for the windowed solver), eventually causing L2 cache pressure. Table 1 confirms this trade-off at 54-bit for `bsgs_dlp_benchmark_cached.c`: performance improves up to $W = 512$ and degrades at $W = 2048$ as the workspace exceeds L2 cache per core.

Table 1: 54-bit solve time vs. window size W for `bsgs` and `bsgs_zaddsub`. $\ell_1 = 31$, 10 threads, seed=12345, 20 trials, Apple M1 Pro. Results differ from Table 2 which averages over 10 seeds.

W	<code>bsgs</code>	<code>bsgs_zaddsub</code>
1	5291 ms	1320 ms
64	1057 ms	625 ms
256	826 ms	298 ms
512	595 ms	219 ms
1024	801 ms	245 ms
2048	801 ms	267 ms

We use $W = 512$ throughout for all windowed solver benchmarks.

5.3 On-Machine Comparison

Table 2 and Table 3 give the complete same-hardware comparison across all implementations on two machines. All implementations solved all trials correctly across all bit sizes and seeds. Two benchmark modes are reported: random trials and worst-case ($m = 2^\ell - 1$, fully deterministic). The reported execution time excludes the one-time baby table T'_1 build and T'_2 precomputation for Tang et al. and `+Parallel`, as both are independent of P_m and amortised across solves.

5.3.1 Apple M1 Pro (32 GB RAM, 10 cores)

Table 2 reports results on the Apple M1 Pro with 1000 trials at 54-bit, 500 at 58-bit, 300 at 63-bit, and 150 at 64-bit per seed. Worst-case results are averaged over 10 trials.

54 and 58-bit. At these ranges, all four implementations are feasible since T'_2 fits in memory (0.34 GB at 54-bit, 5.91 GB at 58-bit for the signed range). `+Parallel` achieves the fastest average solve time at both 54-bit (154 ms, $1.34\times$ over Tang et al.) and 58-bit (2.51 sec, $1.37\times$ over Tang et al.) by eliminating the sequential Phase 1 bottleneck of FastECDLP. At 54-bit, Tang et al. achieves the fastest worst-case time

Table 2: On-machine comparison: Apple M1 Pro, 32 GB RAM, 10 cores (8 P-cores + 2 E-cores), `libsecp256k1`. Avg = average across 10 seeds (1000 trials at 54-bit, 500 trials at 58-bit, 300 trials at 63-bit, 150 trials at 64-bit). Max = worst-case solve ($m = 2^\ell - 1$, 50 trials at 54-bit, 10 trials at 58/63/64-bit). FastECDLP variants use signed range (T'_2 halved).

bits	ℓ_1	Tang et al.		+Parallel		+Windowed		+Co-Z	
		avg	max	avg	max	avg	max	avg	max
54	31	207 ms	330 ms	154 ms	366 ms	242 ms	521 ms	198 ms	377 ms
58	31	3.45 s	16.58 s	2.51 s	11.06 s	3.71 s	7.49 s	2.65 s	5.50 s
63	31	infeasible ($T'_2 = 189$ GB)				118.8 s	242.3 s	80.8 s	183.0 s
64	31	infeasible ($T'_2 = 378$ GB)				240.6 s	496.7 s	182.7 s	354.8 s

(330 ms) since the precomputed T'_2 makes the search fully deterministic once the table is loaded. At 58-bit, +Co-Z achieves the fastest worst-case time (5.50 sec vs 7.49 sec for +Windowed, $1.36\times$ faster).

63 and 64-bit. At these ranges, all FastECDLP variants are infeasible since T'_2 requires 189 GB (63-bit) and 378 GB (64-bit), far exceeding ordinary workstation capacity. +Windowed is the first approach to make these ranges practical by eliminating T'_2 entirely, completing 63-bit in 118.8 sec (avg) and 64-bit in 240.6 sec (avg). +Co-Z further improves upon this, achieving 80.8 sec at 63-bit ($1.47\times$ faster) and 182.7 sec at 64-bit ($1.32\times$ faster).

Worst-case vs average-case. The gap between average and worst-case times matches the theoretical expectation: random plaintexts are uniformly distributed across $[0, 2^\ell)$, so the expected giant-step depth is about half the maximum, giving a ratio of $\approx 2\times$. For +Co-Z this holds at 54-bit (198 ms vs 377 ms, $1.9\times$), 58-bit (2.65 sec vs 5.50 sec, $2.1\times$), 63-bit (80.8 sec vs 183.0 sec, $2.3\times$) and 64-bit (182.7 sec vs 354.8 sec, $1.9\times$). Tang et al. is the exception at 54-bit, where the worst case of 330 ms is only $1.6\times$ the average of 207 ms: the denominators are computed and batch-inverted over the whole of T'_2 before any lookup takes place, so that cost is paid regardless of where the solution lies and only the subsequent table search terminates early.

5.3.2 Intel Core Ultra 9 285K (64 GB RAM, 24 cores)

Table 3 reports results on the Intel Core Ultra 9 285K with 100 trials per seed (1000 total) at all bit sizes. Worst-case results are averaged over 50 runs. Turbo Boost is disabled for reproducibility.

Table 3: On-machine comparison: Intel(R) Core(TM) Ultra 9 285K, 64 GB RAM, 24 cores (8 P-cores + 16 E-cores), Turbo Boost is disabled, `libsecp256k1`. Avg = average across 1000 random trials (10 random seeds, 100 random trials generated per seed). Max = worst-case solve ($m = 2^\ell - 1$, avg. over 50 runs). FastECDLP variants use signed range (T'_2 halved).

bits	ℓ_1	Tang et al.		+Parallel		+Windowed		+Co-Z	
		avg	max	avg	max	avg	max	avg	max
54	31	260 ms	298 ms	89 ms	131 ms	127 ms	187 ms	115 ms	180 ms
58	31	3.80 s	4.40 s	1.40 s	2.10 s	1.99 s	3.34 s	1.71 s	2.97 s
63	31	infeasible ($T'_2 = 189$ GB)				61 s	103 s	53 s	93 s
64	31	infeasible ($T'_2 = 378$ GB)				144 s	204 s	105 s	187 s

54 and 58-bit. At these ranges, all four implementations are feasible since T'_2 fits in memory. +Parallel achieves the fastest average solve time at both 54-bit (89 ms, $2.9\times$ over Tang et al.) and 58-bit (1.40 sec, $2.7\times$ over Tang et al.). +Co-Z achieves competitive performance at 54-bit (115 ms) and 58-bit (1.71 sec) while requiring no T'_2 .

63 and 64-bit. At these ranges, all FastECDLP variants are infeasible since T'_2 requires 189 GB (63-bit) and 378 GB (64-bit). **+Windowed** is the first approach to make these ranges practical, completing 63-bit in 61 sec (avg) and 64-bit in 144 sec (avg). **+Co-Z** further improves this, achieving 53 sec at 63-bit (1.15 $\times$ faster) and 105 sec at 64-bit (1.37 $\times$ faster).

Worst-case vs average-case. The ratio between average and worst-case times is consistent with the theoretical expectation of $\approx 2\times$ across all bit sizes. On the Intel workstation at 54-bit, **+Co-Z** shows 115 ms average vs 180 ms worst-case (1.57 $\times$), and at 58-bit 1.71 sec vs 2.97 sec (1.74 $\times$). At 63-bit, 53 sec vs 93 sec (1.75 $\times$), and at 64-bit 105 sec vs 187 sec (1.78 $\times$). The slightly smaller ratio compared to the theoretical 2 $\times$ reflects the 24-thread parallelism. Each thread covers a smaller portion of the giant-step range, reducing variance between random and worst-case solve times.

Appendix B outlines several tricks/techniques for further optimization. The implementation of these techniques is currently in progress, and we expect to report the resulting performance improvements in a future version of this work.

6 Conclusion

We have presented three algorithmic contributions and one practical engineering contribution to the Baby-Step Giant-Step framework for small ECDLP on **secp256k1**. As a practical engineering contribution, we parallelise Phase 1+2 (denominator computation and batch inversion) across threads, achieving up to a 2.9 $\times$ speedup over Tang et al.’s sequential baseline. As algorithmic contributions, we introduce a Jacobian coordinate loop that advances the giant-step point without a per-step inversion and removes the requirement for the precomputed affine giant-step table T'_2 ; windowed batch inversion, which accumulates W Jacobian points and batch-inverts their Z -coordinates in one shot, reducing per-thread working memory from $O(2^{\ell_2})$ to $O(W) \approx 50$ KB regardless of bit size; and Co-Z arithmetic, which derives a forward and a backward giant step from a shared Z -coordinate at **15M + 4S** per iteration, below the **16M + 6S** of the two independent mixed additions it replaces. Together, the three algorithmic contributions open the door to full-range decryption: T'_2 of Tang et al. requires 5.91 GB at 58-bit, 189 GB at 63-bit and 378 GB at 64-bit, making it infeasible on a workstation, whereas our complete solver requires only $O(W) \approx 50$ KB of working memory per thread. On-machine comparison with a faithful re-implementation of FastECDLP on the same hardware shows that our implementations outperform the Tang et al. baseline on average at every bit size where that baseline can be instantiated. At 54-bit ($\ell_1 = 31$), our complete solver averages 115 ms against 260 ms for the baseline on the Intel machine (2.3 $\times$) and 198 ms against 207 ms on the Apple M1 Pro, in both cases without any T'_2 storage; the parallel Phase 1+2 variant, which retains T'_2 , is faster still at 89 ms and 154 ms respectively. At 58-bit, our complete solver averages 1.71 sec against 3.80 sec (Intel) and 2.65 sec against 3.45 sec (Apple M1 Pro), and improves the worst case from 16.58 sec to 5.50 sec on the latter. At 63-bit and 64-bit the baseline cannot be instantiated at all, since T'_2 would occupy 189 GB and 378 GB; our complete solver averages 53 sec and 105 sec on the Intel machine and 81 sec and 183 sec on the Apple M1 Pro, directly overcoming the memory limitation identified by Tang et al. [30]. Appendix B describes two further arithmetic optimizations, an inversion-chaining scheme and a co- Z Jacobian addition, that lower the per-iteration operation count below that of the solver evaluated here.

References

- [1] Jacques M Bahi, Christophe Guyeux, and Abdallah Makhoul. Efficient and robust secure aggregation of encrypted data in sensor networks. In *2010 Fourth International Conference on Sensor Technologies and Applications*, pages 472–477. IEEE, 2010.
- [2] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database: Jacobian coordinates with $a = 0$ for short Weierstrass curves. <https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html>, 2007. Accessed: 2026-09-03.

- [3] Daniel J Bernstein and Tanja Lange. Computing small discrete logarithms faster. In *International Conference on Cryptology in India*, pages 317–338. Springer, 2012.
- [4] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards Privacy in a Smart Contract World. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.
- [5] Vitalik Buterin. A Next-Generation Smart Contract and Decentralized Application Platform. Ethereum White Paper, 2014. URL: <https://ethereum.org/en/whitepaper/>.
- [6] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018. doi:10.1109/SP.2018.00020.
- [7] Murat Cenk, Aanchal Malhotra, and Joseph A Akinyele. Confidential Transfers for Multi-Purpose Tokens on the XRP Ledger. *IEEE Access*, 2026.
- [8] Panagiotis Chatzigiannis, Konstantinos Chalkias, and Valeria Nikolaenko. Homomorphic decryption in blockchains via compressed discrete-log lookup tables. In *International Workshop on Data Privacy Management*, pages 328–339. Springer, 2021.
- [9] David Chaum and Torben Pryds Pedersen. Wallet Databases with Observers. In *Annual international cryptology conference*, pages 89–105. Springer, 1992.
- [10] Yu Chen, Xuecheng Ma, Cong Tang, and Man Ho Au. PGC: Decentralized Confidential Payment System with Auditability. In *European symposium on research in computer security*, pages 591–610. Springer, 2020.
- [11] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. In *Advances in Cryptology — EUROCRYPT ’97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 1997.
- [12] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and Secure Comparison for On-Line Auctions. In *Australasian conference on information security and privacy*, pages 416–430. Springer, 2007.
- [13] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.
- [14] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. VF2Boost: Very Fast Vertical Federated Gradient Boosting for Cross-Enterprise Learning,. In *Proceedings of the 2021 International Conference on Management of Data*, pages 563–576, 2021. URL: <https://api.semanticscholar.org/CorpusID:235474191>.
- [15] Steven D Galbraith, Ping Wang, and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with Improved Baby-step Giant-step Algorithm. *Advances in Mathematics of Communications*, 11(3), 2017.
- [16] Weichao Gao, Wei Yu, Fan Liang, William Grant Hatcher, and Chao Lu. Privacy-Preserving Auction for Big Data Trading Using Homomorphic Encryption. *IEEE Transactions on Network Science and Engineering*, 7(2):776–791, 2018.
- [17] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [18] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
- [19] Ziwen Liao and Fangguo Zhang. Accelerating Baby-step Giant-step Algorithm for Solving ECDLP Using Elliptic Divisibility Sequences. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):518–540, 2026.

- [20] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious Neural Network Predictions via MiniONN transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, 2017.
- [21] Shunli Ma, Yi Deng, Debiao He, Jiang Zhang, and Xiang Xie. An Efficient NIZK Scheme for Privacy-Preserving Transactions over Account-Model Blockchain. *IEEE Transactions on Dependable and Secure Computing*, 18(2):641–651, 2020.
- [22] Nicolas Meloni. New Point Addition Formulae for ECC Applications. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, pages 189–201, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [23] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.
- [24] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.
- [25] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- [26] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.
- [27] Daniel Sandler, Kyle Derr, and Dan S Wallach. VoteBox: A Tamper-evident, Verifiable Electronic Voting System. In *USENIX Security Symposium*, volume 4, page 87, 2008.
- [28] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Math. Soc., 1971*, volume 20, pages 415–440, 1971.
- [29] Solana Foundation. Confidential Transfer. Solana Documentation. Accessed: 2026-08-19. URL: <https://solana.com/docs/tokens/extensions/confidential-transfer>.
- [30] Fei Tang, Guowei Ling, Chaochao Cai, Jinyong Shan, Xuanqi Liu, Peng Tang, and Weidong Qiu. Solving small exponential ECDLP in EC-based additively homomorphic encryption and applications. *IEEE Transactions on Information Forensics and Security*, 18:3517–3530, 2023.
- [31] Osman Ugus, Dirk Westhoff, Ralf Laue, Abdulhadi Shoufan, and Sorin A Huss. Optimized Implementation of Elliptic Curve Based Additive Homomorphic Encryption for Wireless Sensor Networks. *arXiv preprint arXiv:0903.3900*, 2009.
- [32] Gavin Wood et al. Ethereum: A secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
- [33] Zhenjie Xie, Shengli Liu, Yao Zhang, and Youqiang Luo. FastGTDLP: efficient algorithm for small-exponent discrete logarithm problem over group GT. *Cybersecurity*, 9(1):80, 2026.
- [34] XRPL Foundation. XRP Ledger Documentation. XRPL Documentation. Accessed: 2026-08-19. URL: <https://xrpl.org/docs/>.
- [35] XRPL Foundation. XLS-96: Confidential Transfers for the XRP Ledger. XRPL Standards Proposal, 2024.
- [36] Yu Zhan, Wei Zhao, Chaoxi Zhu, Zhen Zhao, Ning Yang, and Baocang Wang. Efficient Electronic Voting System Based on Homomorphic Encryption. *Electronics*, 13(2):286, 2024.
- [37] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. BatchCrypt: Efficient homomorphic encryption for Cross-Silo federated learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 493–506. USENIX Association, July 2020. URL: <https://www.usenix.org/conference/atc20/presentation/zhang-chengliang>.

- [38] Chong Zhang, Xiaojun Zhang, Xingchun Yang, Bingyun Liu, Yuan Zhang, and Rang Zhou. Poisoning attacks resilient privacy-preserving federated learning scheme based on lightweight homomorphic encryption. *Information Fusion*, 121:103131, 2025.

A Windowed-Batch Inversion BSGS Adapting to Signed Range using Conjugate co-Z point addition (ZADDC)

In this section, we present an alternative algorithmic structure for windowed-batch inversion BSGS over the signed range $[-2^{\ell-1}, 2^{\ell-1})$. In the structure proposed in Section 3.3, at iteration j , we compute $Q_{sub}^j \leftarrow Q_{sub}^{j-1} - S$ and $Q_{add}^j \leftarrow Q_{add}^{j-1} + S$, where $S = M \cdot G$ in affine coordinate and both Q_{add} and Q_{sub} share a common Z -coordinate. Alternatively, they can be computed directly as $Q_{sub}^j \leftarrow P_m - j \cdot S$ and $Q_{add}^j \leftarrow P_m + j \cdot S$, similar to FastECDLP. However, unlike FastECDLP, we do not precompute $j \cdot S = j \cdot M \cdot G$. To achieve this, we utilize conjugate co- Z point addition (ZADDC) [17] as given in Algorithm 11 alongside a variant of mixed Jacobian-affine addition (JacAffADDUPDATE), shown in Algorithm 12.

Assuming that in the first iteration P_m and S_{jac} (the Jacobian representation of the affine point S) share a common Z -coordinate, we can compute $(Q_{add}, Q_{sub}) \leftarrow \text{ZADDC}(P_m, S_{jac})$, which costs $6M + 3S$ using Algorithm 11. Crucially, this preserves a shared Z -coordinate for both Q_{add} and Q_{sub} , consistent with our first approach in Section 3.3. In subsequent iterations, we calculate the next giant step $S_{jac} \leftarrow S_{jac} + S$ and rescale P_m to ensure that S_{jac} and P_m maintain a common Z -coordinate. To this end, we propose a mixed Jacobian-affine addition with update (Algorithm 12), which extends standard mixed Jacobian-affine addition by taking an additional input. The output Z -coordinate of S_{jac} is computed as the input Z -coordinate multiplied by H_a ; thus, to scale P_m , we compute $\tilde{X} \leftarrow XH_a^2$ and $\tilde{Y} \leftarrow YH_a^3$. Given that H_a^2 and H_a^3 are already computed, rescaling P_m costs only $2M$. Consequently, this mixed Jacobian-affine addition with update (Algorithm 12) costs $10M + 3S$.

Algorithm 11 Conjugate co- Z point addition (ZADDC) [17]

Input: $P_1 = (X_1 : Y_1 : Z)$, and $P_2 = (X_2 : Y_2 : Z)$

Output: $(P_a, P_s) \leftarrow \text{ZADDC}(P_1, P_2)$ where $P_a \leftarrow P_1 + P_2 = (X_a : Y_a : Z_c)$ and $P_s \leftarrow P_1 - P_2 = (X_s : Y_s : Z_c)$

$$\begin{aligned}
C &\leftarrow (X_1 - X_2)^2, \\
W_1 &\leftarrow X_1 C, \quad W_2 \leftarrow X_2 C, \\
D_a &\leftarrow (Y_1 - Y_2)^2, \quad D_s \leftarrow (Y_1 + Y_2)^2, \\
A_1 &\leftarrow Y_1(W_1 - W_2), \\
X_a &\leftarrow D_a - W_1 - W_2, \quad Y_a \leftarrow (Y_1 - Y_2)(W_1 - X_a) - A_1, \\
X_s &\leftarrow D_s - W_1 - W_2, \quad Y_s \leftarrow (Y_1 + Y_2)(W_1 - X_s) - A_1, \\
Z_c &\leftarrow Z(X_1 - X_2)
\end{aligned}$$

BSGS for signed range using conjugate co- Z point addition. Algorithm 13 adapts Algorithm 10 by utilizing the ZADDC and JacAffADDUPDATE function.

The cost of the worst-case where BSGS requires the maximum number of iterations (2^{ℓ_2-1}) is as follows:

$$\begin{aligned}
\text{Cost} &= \frac{2^{\ell_2}}{2} \times \left((6M + 3S) + \frac{1}{W}(11 + 3(W-1)M) + (2M + 1S) + (10M + 3S) \right) \\
&\approx \frac{2^{\ell_2}}{2} \times \left((21M + 7S) + \frac{11}{W} \right)
\end{aligned}$$

Algorithm 12 Mixed Jacobian–affine addition with update (JacAffADDUPDATE)

Input: $P_1 = (X_1 : Y_1 : Z)$, $P_2 = (X_2 : Y_2 : Z)$ and $P = (x, y)$

Output: $(P_a, \tilde{P}_2) \leftarrow \text{JacAffADDUPDATE}(P_1, P_2, P)$ where $P_a \leftarrow P_1 + P = (X_a : Y_a : Z_c)$ and $\tilde{P}_2 \leftarrow (\tilde{X}_2 : \tilde{Y}_2 : Z_c) = (\lambda^2 X_2 : \lambda^3 Y_2 : Z_c)$

$$\begin{aligned} U &\leftarrow xZ^2, \quad V \leftarrow yZ^3, \\ H_a &\leftarrow U - X_1, \quad D_a \leftarrow V - Y_1, \\ X_a &\leftarrow D_a^2 - H_a^3 - 2X_1H_a^2, \\ Y_a &\leftarrow D_a(X_1H_a^2 - X_a) - Y_1H_a^3, \\ Z_c &\leftarrow ZH_a, \\ \tilde{X}_2 &\leftarrow H_a^2X_2, \quad \tilde{Y}_2 \leftarrow H_a^3Y_2. \end{aligned}$$

Therefore, the signed range algorithm using Algorithm 13 requires $\frac{2\ell_2}{2} \times \left((3M + 1S) + \frac{11}{W} \right)$ operations less than the unsigned range algorithm for the worst-case, which is equivalent to up to 12.5% improvement.

B Further Optimization

Below we describe two additional techniques to reduce the arithmetic cost of the giant step. These are then applied to the unsigned and signed range versions of our BSGS algorithms presented earlier.

B.1 Inversion chain

In Algorithm 7, we treat the Z -coordinates as independent; therefore, we use ‘Montgomery’ batch inversion. For a batch of size W , the inversion costs $11 + 3(W-1)M$. Since we are interested in $(Z^{-1})^2$ to rescale X to its affine coordinate, we compute $(Z^{-1})^2$, which costs $1S$. The total cost per iteration before rescaling X to its affine coordinate is $\approx 3M + 1S$. However,

$$Q_j \leftarrow \text{JacAffADD}(Q_{j-1}, \text{neg}S)$$

Therefore, Z -coordinates are created a chain as shown in the mixed Jacobian–affine addition (Algorithm 1),

$$\begin{aligned} Z_j &\leftarrow Z_{j-1} \cdot H_j \\ Z_{j-1}^{-1} &= Z_j^{-1} \cdot H_j \\ (Z_{j-1}^{-1})^2 &= (Z_j^{-1})^2 \cdot H_j^2 \end{aligned}$$

For a batch of size W , by keeping track of H_j^2 , we can compute the chain inversion as follows: compute the inverse of Z_W to get Z_W^{-1} , then square it to get $(Z_W^{-1})^2$. For each iteration, we compute $(Z_{j-1}^{-1})^2 \leftarrow (Z_j^{-1})^2 \cdot H_j^2$. We refer to this procedure as **BatchInversionChain**. For a batch of size W , **BatchInversionChain** costs $11 + (W-1)M + 1S$. The total amortized cost per iteration, before rescaling X to its affine coordinate, is $\approx 1M$, saving $\approx 2M + 1S$ per iteration.

B.2 Co-Z Jacobian addition

Mixed Jacobian–affine addition (Algorithm 1) costs $8M + 3S$. Internally, it rescales the affine point P_2 so that P_1 and P_2 share the same Z -coordinate (Z_1), which costs $3M + 1S$. Furthermore, it rescales P_1 so that P_1 and P_3 share Z_3 at no extra cost. Instead of using the affine coordinates of P_2 , we use its Jacobian coordinates as follows:

Algorithm 13 Windowed Batch Inversion BSGS for $m \in [-2^{\ell-1}, 2^{\ell-1})$

Input: $P_m = (x_m : y_m : 1)$, baby-step table T'_1 , step $S = M \cdot G = (x_S, y_S)$, window W **Output:** m or $\perp$

```
1: if  $(i, ok) \leftarrow \text{Search}(T'_1, x_m)$ ;  $ok$  then
2:   return first of  $\{i, -i\}$  satisfying  $\cdot G = P_m$ 
3: end if
4:  $Q_{sub}, Q_{add}, S_{jac} \leftarrow (x_S : y_S : 1)$ ;  $buf \leftarrow []$ 
5: for  $j = 1$  to  $2^{\ell_2-1}$  do
6:    $(Q_{add}, Q_{sub}) \leftarrow \text{ZADDC}(P_m, S_{jac})$  ▷ 6M + 3S
7:   if  $Q_{sub} = \mathcal{O}$  then
8:     return  $m = j \cdot M$ 
9:   end if
10:  if  $Q_{add} = \mathcal{O}$  then
11:    return  $m = -j \cdot M$ 
12:  end if
13:   $buf.append(((X_{add}, X_{sub}, Z), j))$ 
14:  if  $|buf| = W$  or  $j = 2^{\ell_2-1}$  then ▷ 1I + 3(W-1)M
15:     $\{Z_k^{-1}\} \leftarrow \text{BatchInvert}(\{Z_k\})$ 
16:    for each  $((X_{add_k}, X_{sub_k}, Z_k), j_k)$  in  $buf$  do
17:       $x_k \leftarrow X_{sub_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M + 1S
18:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
19:      if  $ok$  then
20:        return  $m \in \{j_k \cdot M + i, j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
21:      end if
22:       $x_k \leftarrow X_{add_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M
23:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
24:      if  $ok$  then
25:        return  $m \in \{-j_k \cdot M + i, -j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
26:      end if
27:    end for
28:     $buf \leftarrow []$ 
29:  end if
30:   $(S_{jac}, P_m) \leftarrow \text{JacAffADDUPDATE}(S_{jac}, P_m, S)$  ▷ 10M + 3S
31: end for
```

Algorithm 14 Co-Z Jacobian addition (ZADDU) [22]

Input: $P_1 = (X_1 : Y_1 : Z)$ in Jacobian, $P_2 = (X_2, Y_2 : Z)$ in Jacobian**Output:** $(P_3, \tilde{P}_1, H^2) \leftarrow \text{ZADDU}(P_1, P_2)$ where $P_3 \leftarrow P_1 + P_2 = (X_3 : Y_3 : Z_c)$ in Jacobian and $\tilde{P}_1 = (\tilde{X}_1 : \tilde{Y}_1, Z_c) = (X_1 H^2 : Y_1 H^3 : ZH)$

```
function ZADDU( $P_1, P_2$ )
   $U_2 \leftarrow X_2,$   $V_2 \leftarrow Y_2,$ 
   $H \leftarrow U_2 - X_1,$   $D \leftarrow V_2 - Y_1,$ 
   $Z_c \leftarrow ZH,$ 
   $\tilde{X}_1 \leftarrow X_1 H^2,$ 
   $\tilde{Y}_1 \leftarrow Y_1 H^3,$ 
   $X_3 \leftarrow D^2 - H^3 - 2\tilde{X}_1,$ 
   $Y_3 \leftarrow D(X_1 H^2 - X_3) - \tilde{Y}_1.$ 
end function
```

The cost of ZADDU, as described in Alg. 14, is $5M + 2S$, saving $3M + 1S$ compared to JacAffADD.

In BSGS, instead of

$$(Q_j, H^2) \leftarrow \text{JacAffADD}(Q_{j-1}, \text{neg}S)$$

we can use:

$$(Q_j, \text{neg}S, H^2) \leftarrow \text{ZADDU}(\text{neg}S, Q_{j-1})$$

B.3 New Windowed-Batch Inversion BSGS using BatchInversionChain and ZADDU

In Alg. 15, we apply BatchInversionChain and ZADDU procedures, directly replacing the corresponding BatchInvert and JacAffADD, for BSGS with $m \in [0, 2^\ell)$.

Algorithm 15 New Windowed-Batch Inversion BSGS for $m \in [0, 2^\ell)$ using BatchInversionChain and ZADDU

Input: $P_m = (x_m : y_m : 1)$, baby-step table T'_1 , step $S = M \cdot G = (x_S : y_S : 1)$, window W

Output: m or $\perp$

```

1:  $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_m)$ 
2: if  $ok$  then
3:   return  $m \leftarrow i$ 
4: end if
5:  $Q \leftarrow P_m; \text{neg}S \leftarrow -S, \text{ buf} \leftarrow []$ 
6: for  $j = 1$  to  $2^{\ell_2}$  do
7:    $Q \leftarrow \text{JacAffADD}(Q, \text{neg}S)$   $\triangleright 8M + 3S$ 
8:    $(Q, \text{neg}S, H^2) \leftarrow \text{ZADDU}(\text{neg}S, Q)$   $\triangleright 5M + 2S$ 
9:   if  $Q = \mathcal{O}$  then  $\triangleright P_m = j \cdot M \cdot G$ 
10:    return  $m \leftarrow j \cdot M$ 
11:  end if
12:   $\text{buf.append}((Q, j))$ 
13:   $\text{buf.append}(((X, H^2), j))$ 
14:  if  $|\text{buf}| = W$  or  $j = 2^{\ell_2}$  then
15:     $\{Z_k^{-1}\} \leftarrow \text{BatchInvert}(\{Z_k\})$   $\triangleright 1I + 3(W-1)M$ 
16:     $\{(Z_k^{-1})^2\} \leftarrow \text{BatchInversionChain}(Z_W, \{H_k^2\})$   $\triangleright 1I + (W-1)M + 1S$ 
17:    for each  $((X_k, H_k^2), j_k)$  in  $\text{buf}$  do
18:       $x_k \leftarrow X_k \cdot (Z_k^{-1})^2 \bmod p$   $\triangleright 1M + 1S$ 
19:       $x_k \leftarrow X_k \cdot (Z_k^{-1})^2 \bmod p$   $\triangleright 1M$ 
20:       $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
21:      if  $ok$  then
22:        return  $m \in \{j_k \cdot M + i, j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
23:      end if
24:    end for
25:     $\text{buf} \leftarrow []$ 
26:  end if
27: end for

```

Cost Analysis. The cost of the worst-case where BSGS requires the maximum number of iterations (2^{ℓ_2}) is as follows:

$$\begin{aligned}
\text{Cost}_{\text{new}} &= 2^{\ell_2} \times \left((5M + 2S) + \frac{1}{W}(1I + (W-1)M + 1S) + (1M) + 1L \right) \\
&\approx 2^{\ell_2} \times \left((7M + 2S) + \frac{1I}{W} + 1L \right) \\
&\approx \frac{2^{\ell_2}}{2} \times \left((14M + 4S) + 2 \cdot \frac{1I}{W} + 2L \right)
\end{aligned}$$

Therefore, the new version requires $\frac{2^{\ell_2}}{2} \times ((10M + 4S))$ operations less than the current version (Algorithm 7), which is equivalent to up to 40% improvement.

B.4 New Signed Range BSGS

The aforementioned BatchInversionChain procedure and a variant of ZADDU for combined Jacobian addition and subtraction (ZADDSUBU), corresponding to JacAffADDSUB, can also be applied to the signed range of BSGS, i.e., Algorithm 10. See Alg. 16 below.

Algorithm 16 Windowed-Batch Inversion BSGS for $m \in [-2^{\ell-1}, 2^{\ell-1})$ using BatchInversionChain and ZADDSUBU

Input: $P_m = (x_m : y_m : 1)$, baby-step table T'_1 , step $S = M \cdot G = (x_S : y_S : 1)$, window W

Output: m or $\perp$

```

1:  $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_m)$ 
2: if  $ok$  then
3:   return  $m \in \{i, -i\}$  s.t.  $m \cdot G = P_m$ 
4: end if
5:  $Q_{sub} \leftarrow P_m, Q_{add} \leftarrow P_m; \text{buf} \leftarrow []$ 
6: for  $j = 1$  to  $2^{\ell_2-1}$  do
7:    $(Q_{add}, Q_{sub}) \leftarrow \text{JacAffADDSUB}(Q_{add}, Q_{sub}, S)$  ▷ 15M + 4S
8:    $(Q_{add}, Q_{sub}, S, H^2) \leftarrow \text{ZADDSUBU}(Q_{add}, Q_{sub}, S)$  ▷ 11M + 3S
9:   if  $Q_{sub} = \mathcal{O}$  then
10:    return  $m \leftarrow j \cdot M$ 
11:   end if
12:   if  $Q_{add} = \mathcal{O}$  then
13:    return  $m \leftarrow -j \cdot M$ 
14:   end if
15:    $\text{buf.append}(((X_{add}, X_{sub}, Z), j))$ 
16:    $\text{buf.append}(((X_{add}, X_{sub}, H^2), j))$ 
17:   if  $|\text{buf}| = W$  or  $j = 2^{\ell_2-1}$  then
18:      $\{Z_k^{-1}\} \leftarrow \text{BatchInvert}(\{Z_k\})$  ▷ 11 + 3(W-1)M
19:      $\{(Z_k^{-1})^2\} \leftarrow \text{BatchInversionChain}(Z_W, \{H_k^2\})$  ▷ 11 + (W-1)M + 1S
20:     for each  $((X_{add_k}, X_{sub_k}, Z_k), j_k)$  in  $\text{buf}$  do
21:        $x_k \leftarrow X_{sub_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M + 1S
22:        $x_k \leftarrow X_{sub_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M
23:        $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
24:       if  $ok$  then
25:         return  $m \in \{j_k \cdot M + i, j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
26:       end if
27:        $x_k \leftarrow X_{add_k} \cdot (Z_k^{-1})^2 \bmod p$  ▷ 1M
28:        $(i, ok) \leftarrow \text{SEARCH}(T'_1, x_k)$ 
29:       if  $ok$  then
30:         return  $m \in \{-j_k \cdot M + i, -j_k \cdot M - i\}$  s.t.  $m \cdot G = P_m$ 
31:       end if
32:     end for
33:    $\text{buf} \leftarrow []$ 
34: end if
35: end for

```

Cost Analysis. The cost of the worst-case where BSGS requires the maximum number of iterations (2^{ℓ_2-1}) is as follows:

$$\begin{aligned} Cost_{new} &= \frac{2^{\ell_2}}{2} \times \left((11M + 3S) + \frac{1}{W}(1I + (W-1)M + 1S) + (2M) + 2L \right) \\ &\approx \frac{2^{\ell_2}}{2} \times \left((14M + 3S) + \frac{11}{W} + 2L \right) \end{aligned}$$

Therefore, the new version requires $\frac{2^{\ell_2}}{2} \times ((6M + 2S))$ operations less than the current version (Algorithm 10), which is equivalent to up to 32% improvement.