Solving Small Exponential ECDLP in EC-based Additively Homomorphic Encryption and Applications

Fei Tang, Guowei Ling, Chaochao Cai, Jinyong Shan, Xuanqi Liu, Peng Tang, Weidong Qiu

***Abstract*—Additively Homomorphic Encryption (AHE) has been widely used in various applications, such as federated learning, blockchain, and online auctions. Elliptic Curve (EC) based AHE has the advantages of efficient encryption, homomorphic addition, scalar multiplication algorithms, and short ciphertext length. However, EC-based AHE schemes require solving a small exponential Elliptic Curve Discrete Logarithm Problem (ECDLP) when running the decryption algorithm, i.e., recovering the plaintext $m \in \{0,1\}^\ell$ from $m * G$. Therefore, the decryption of EC-based AHE schemes is inefficient when the plaintext length $\ell > 32$. This leads to people being more inclined to use RSA-based AHE schemes rather than EC-based ones.**

This paper proposes an efficient algorithm called FastECDLP for solving the small exponential ECDLP at 128-bit security level. We perform a series of deep optimizations from two points: computation and memory overhead. These optimizations ensure efficient decryption when the plaintext length ℓ is as long as possible in practice. Moreover, we also provide a concrete implementation and apply FastECDLP to some specific applications. Experimental results show that FastECDLP is far faster than the previous works. For example, the decryption can be done in 0.35 ms with a single thread when $\ell = 40$, which is about 30 times faster than that of Paillier. Furthermore, we experiment with ℓ from 32 to 54, and the existing works generally only consider $\ell \le 32$. The decryption only require 1 second with 16 threads when $\ell = 54$. In the practical applications, we can speed up model training of existing vertical federated learning frameworks by 4 to 14 times. At the same time, the decryption efficiency is accelerated by about 140 times in a blockchain financial system (ESORICS 2021) with the same memory overhead.

***Index Terms*—ECDLP, additively homomorphic encryption, fast decryption, BSGS, cuckoo hashing.**

I. Introduction

The concept of Homomorphic Encryption (HE) was introduced by Rivest et al. [1] to resolve the privacy-preserving problems of ciphertext computation. HE contains two categories: fully homomorphic encryption [2] and partially homomorphic encryption [3]. Additively Homomorphic Encryption (AHE) is a sort of partially homomorphic encryption that supports addition computation in the ciphertext state. AHE is usually the fundamental core component for building privacy-preserving machine learning frameworks [4], [5], federated learning frameworks [6], [7], blockchain systems [8], [9], and online auction systems [10], [11].

There are two types of mainstream AHE schemes are: EC-based AHE [8], [9], [12]–[14] and RSA-based AHE [3], [10], [15]–[17]. EC-based AHE schemes usually have advantages over RSA-based AHE schemes in terms of efficiency and ciphertext length since a very large RSA modulus n should be chosen for security. For example, n should be set to 3072 bits for 128-bit security level, and it is recommended not to set n to 2048 bits[1] after 2023 [18]. However, EC-based AHE schemes encode the plaintext $m \in \{0,1\}^\ell$ as $m*G$ in order to obtain additive homomorphism. Therefore, we cannot directly get m in EC-based AHE schemes because of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Note that the performance of ECDLP is almost the same as that of EC-based AHE decryption. Thus, existing works [6], [8], [9], [14] usually only consider the case of $\ell \le 32$ for the decryption efficiency. This severely limits the application of EC-based AHE schemes. Consequently, RSA-based AHE schemes are more widely used than EC-based AHE schemes, especially Paillier [3].

TABLE I
RUNTIME AND CORRESPONDING MEMORY OVERHEAD FOR SOLVING ECDLP WITH A SINGLE THREAD AT 128-LEVEL SECURITY. ℓ_1 AND ℓ_2 DENOTE THE PARAMETERS OF THE BSGS ALGORITHM, WHERE $\ell_1 + \ell_2 = \ell$.

ℓ (bits)	ℓ_1	ℓ_2	Runtime (s)	Memory (GB)
20	13	7	0.001	0.0004
24	17	7	0.001	0.0078
28	21	7	0.001	0.125
32	24	8	0.002	1.03
36	27	9	0.003	8.05
40	27	13	0.049	8.05
44	27	17	0.883	8.05
48	27	21	**16.202**	8.11
52	27	25	**291.670**	9.07
54	27	27	**1312.519**	11.91

Although there are also many methods [19]–[24] for solving ECDLP, they are only efficient when plaintext length ℓ is short (i.e., $\ell \le 32$) [8], [9], [14]. For example, we use the Baby-Step Giant-Step (BSGS) algorithm [19], [24] to recover $m \in \{0,1\}^\ell$ from $m*G$ and give the runtime and memory overhead

[1]In industry, n is sometimes set to 2048 bits for efficiency, but this only corresponds to 112-bit security level.

Fei Tang and Guowei Ling are with Chongqing University of Posts and Telecommunications, China, Chongqing, 400065.
E-mail: tangfei@cqupt.edu.cn; s200201071@stu.cqupt.edu.cn
Chaochao Cai and Jinyong Shan are with Beijing Sudo Technology Co., LTD, China, Beijing, 100083.
Xuanqi Liu is with Tsinghua University, China, Beijing, 100084.
Peng Tang and Weidong Qiu are with Shanghai Jiao Tong University, China, Shanghai, 200240.
This work was supported by the National Key Research and Development Program of China under Grant 2021YFF0704102.

TABLE II
BENCHMARKS OF EXISTING AHE SCHEMES (128-BIT SECURITY LEVEL, AVERAGE OF 1000 TIMES, PLAINTEXT LENGTH ℓ: 48-BIT, $\ell_1 = 27, \ell_2 = 21$, A SINGLE THREAD). Enc AND Enc* DENOTE THE ENCRYPTION ALGORITHMS BEFORE AND AFTER PRECOMPUTATION (I.E., GENERATING ZERO'S CIPHERTEXTS IN ADVANCE), RESPECTIVELY. Dec, HomoAdd, AND ScalarMult DENOTE THE DECRYPTION, HOMOMORPHIC ADDITION, AND SCALAR MULTIPLICATION ALGORITHMS, RESPECTIVELY. NOTE THAT DGK [10] IS A SPECIAL RSA-BASED AHE SCHEME THAT ALSO REQUIRES SOLVING THE SMALL EXPONENTIAL DISCRETE LOGARITHM PROBLEM DURING DECRYPTION PROCESS.

Schemes	Category	Enc (ms)	Enc* (ms)	Dec (ms)	HomoAdd (ms)	ScalarMult (ms)	Ciphertext length (bits)
Exp-ElGamal [12]	EC	0.193	0.03	**16202.13**	0.002	0.22	528
OU [15]	RSA	8.74	0.28	1.33	0.005	0.22	3072
Paillier [3]	RSA	35.68	0.25	9.58	0.02	0.82	6144
BGN [13]	EC	33.51	0.55	**20515.84**	0.016	0.83	264
DGK [10]	RSA	3.50	0.24	**16792.25**	0.006	0.25	3072
JL [16]	RSA	0.46	0.25	2.32	0.005	0.29	3072
Twisted-ElGamal [8]	EC	1.57	0.55	**16195.24**	0.003	0.68	528
Opt-Paillier [17]	RSA	47.18	0.26	6.38	0.02	0.81	6144

in Table I. BSGS is a commonly used method in practice for solving ECDLP [24] due to its stable performance. Note that in choosing the parameters of BSGS, we almost maximized the performance with an acceptable memory overhead for an ordinary PC (i.e., less than 12 GB). As seen from Table I, although the recovery from $m * G$ to m is efficient when $\ell \leq 32$, the performance is unacceptable as ℓ gradually increases. When ℓ is 40, 48, and 54, it costs about 0.05, 16, and 1312 seconds, respectively.

We also evaluate almost all mainstream AHE schemes with a single thread at 128-bit security level and show their performance when $\ell = 48$ in Table II. It shows that the decryption efficiency of EC-based AHE schemes is inefficient compared to that of RSA-based AHE schemes when $\ell > 32$. As a result, even though EC-based AHE schemes have advantages in other aspects, people have to use RSA-based AHE schemes because of the decryption efficiency. For instance, Exp-ElGamal [12], a well-known EC-based AHE scheme, has efficient encryption, homomorphic addition, scalar multiplication algorithms, and a short ciphertext length, according to Table II. However, its poor decryption efficiency is unacceptable if the plaintext m is slightly large since the ECDLP should be solved. Therefore, a host of works [25]–[28] use Paillier [3] rather than Exp-ElGamal [12].

To sum up, the ECDLP limits the application of EC-based AHE schemes. That is, the decryption is inefficient if $\ell > 32$. Our main goal is to improve the decryption efficiency of EC-based AHE schemes when the plaintext length is as long as possible. Note that the plaintext length ℓ is limited since ECDLP is hard to solve when $m \in \{0, 1\}^\ell$ is very large. Even so, the plaintexts to be encrypted are not very large in many application scenarios [6], [8], [9]. Therefore, our work can enable EC-based AHE schemes to be more widely used.

A. Motivations

Since EC-based AHE schemes require solving the small exponent ECDLP during decryption, RSA-based AHE schemes are preferred in the previous works [25]–[29]. As a result, the advantages of EC-based schemes hardly come into play in the AHE scenarios. Therefore, there is a great need for an efficient solution to ensure efficient decryption when the plaintext length is as long as possible. A few existing works [22], [24], [30], [31] also try to speed up the solution of the small exponential ECDLP in EC-based AHE schemes. However, they are still relatively inefficient when $\ell > 32$ at 128-bit security level. Moreover, they are theoretical and provide only a rough estimate of the complexity. At the same time, no adequate experimental results were given. Therefore, it is not easy to evaluate their specific performance when the plaintext length ℓ is longer. In this work, we present a new algorithm called FastECDLP to solve the above problems. We provide a concrete implementation and show some specific application scenarios, including federated learning, blockchain, and online auctions. Furthermore, we also do our best to show as complete experimental results as possible, which can enhance readers' confidence in applying EC-based AHE schemes combined with FastECDLP.

B. Contributions

We summarize the main contributions in this paper as follows.

- We propose an efficient algorithm called FastECDLP for solving the small exponential ECDLP in EC-based AHE schemes. The foundation of FastECDLP is the BSGS algorithm [19] combined with the tree-based Montgomery's trick [32] and cuckoo hashing [33]. In addition, we also perform a series of optimizations about computation and memory overhead.
- We give all experimental results of ℓ from 32 to 54, which demonstrates that FastECDLP is amazing for the decryption efficiency of EC-based AHE schemes. For instance, when $\ell = 40$, the decryption only require about 0.35 ms with a single thread, which is about 4 and 30 times faster than that of OU and Paillier, respectively. We also present some experimental results of applying FastECDLP to various EC-based AHE schemes [8], [12], [13]. It indicates that FastECDLP is generic for any EC-based AHE scheme. Furthermore, the decryption can be done in about 4 seconds with a single thread when $\ell = 54$, while the BSGS algorithm [19], [24] requires about 20 minutes, according to Tables IV and I. Meanwhile, the decryption only costs about 1 second with 16 threads when $\ell = 54$.
- We apply FastECDLP to three specific applications, i.e., federated learning, blockchain, and online auctions. In

federated learning, we can speed up model training of existing frameworks [7], [25], [34], [35] by about 4 to 14 times. In blockchain, compared to [8], we can accelerate the decryption efficiency by about 20 times and reduces memory overhead by about 11 times with the same parameters. Furthermore, if the memory overhead is constant, then we can improve the decryption efficiency by 140 times. If the computation overhead is constant, then we can reduce the memory overhead by 325 times. In online auctions, compared to [10], we expand the range of bids and experiment at a higher security level.

• We have opened the source code of FastECDLP, which can be found at: https://github.com/ShallMate/FastECDLP. This paper is a comprehensive work in solving the small exponential ECDLP in EC-based AHE schemes, including the design of the algorithm, optimization details, code, and applications. Our work can bring renewed attention to EC-based AHE schemes.

II. Related Works

This section briefly reviews the works devoted to solving ECDLP when the exponent is in an interval, i.e., $m \in \{0,1\}^{\ell}$, which is the essence of decryption efficiency for EC-based AHE schemes.

Shanks [19] described the well-known BSGS algorithm runing in time $\mathcal{O}(2^{\ell_2})$ using a table of size $\mathcal{O}(2^{\ell_1})$, where $\ell_1 + \ell_2 = \ell$. Pollard [20] presented the kangaroo method run in time $\mathcal{O}(2^{\ell/2})$ with constant memory overhead. They are the two most commonly used algorithms to solve the ECDLP. Since the average complexity of BSGS is within a constant factor of its worst-case complexity, its performance is better than the kangaroo method in practice [24]. Matsuo et al. [36] showed a variation of BSGS to improve the square–root algorithm. However, [36] is only efficient under the 135-bit prime order curves, which is far from the security requirements of EC-based AHE schemes. Subsequently, Gaudry and Schost [37] gave the parallel version of [36] but still could not overcome the inefficiency for a large prime order. Moreover, Galbraith and Ruprai [21] designed an algorithm based on [20], [37] with smaller computation overhead than previous works. However, they also pointed out that the algorithm is inefficient in practice due to problems with pseudorandom walks going outside the boundaries of the search space and the overhead of handling fruitless cycles. At the same time, the experiments are only completed under a 50-bit prime order curve, which falls far short of the security requirements for EC-based AHE schemes. Bernstein and Lange [22] proposed an algorithm with computation complexity of $\mathcal{O}(2^{\ell/3})$ and memory complexity of $\mathcal{O}(2^{\ell/3})$. It seems to be an excellent computation and memory overhead. However, the complexities are estimated in the particular case of a 48-bit prime order group rather than through theoretical analysis. Gao et al. [38] proposed a solution for ECDLP using MapReduce and parallel collision search in the cloud environment. Although [38] can increase the plaintext length of EC-based AHE schemes by using a distributed database, the communication and database queries result in inferior performance. Galbraith et al. [30] improved the efficiency of BSGS [19] by getting $P_1 + P_2$ when computing $P_1 - P_2$ for two EC points P_1 and P_2. However, it requires three tables of size $\mathcal{O}(2^{\ell/2})$, which is roughly equivalent to 3 times the memory of BSGS. Moreover, [30] did not consider memory optimizations. Therefore, this algorithm is only suitable for scenarios with an very short plaintext length due to the considerable memory overhead, which further limits the application of EC-based AHE schemes. Shafagh et al. [31] described an optimization on BSGS based on the Chinese Remainder Theorem (CRT). They divided the plaintext into three parts and performed encryption, decryption, and homomorphic operations on them, respectively. The computation overhead and the ciphertext length are increased by three times. Although this approach is friendly for the homomorphic addition operation, it will cause overflow when performing the scalar multiplication operation. Therefore, it is also difficult to use it in some applications with many multiplications, such as vertical federated learning [25], [29], [34]. Chatzigiannis et al. [24] reduced the memory overhead of BSGS by compressing coordinates. However, they ignored that the hashmap has much redundancy in practice and did not have any computation overhead optimization.

In conclusion, existing works do not meet the goal of this paper, i.e., efficient decryption of EC-based AHE schemes when the plaintext length is as long as possible, enabling EC-based AHE schemes to be more widely used.

III. Preliminaries

A. Notations

Let GroupGen be a polynomial probabilistic time algorithm that on input the security parameter λ, outputs descriptions of an EC point-group $\mathbb{G}$ of prime order N, and a generator G of the group, where G is the base point of an elliptic curve $\mathbb{E}_p : y^2 = x^3 + ax + b \mod p$ on finite field $\mathbb{F}_p$ and p is a large prime number. Other notations are as follows.

• m denotes the plaintext. Given a plaintext m, $[\![m]\!]$ denotes its ciphertext of AHE schemes. P_m denotes $m * G$, where '$*$' denotes the scalar multiplication on $\mathbb{E}_p$.

• $[\alpha, \beta]$ denotes the set $\{\alpha, \alpha+1, \cdots, \beta\}$.

• We use $\ell = \ell_1 + \ell_2$ to control the plaintext length ℓ and balance the computation and memory overhead by adjusting ℓ_1 and ℓ_2.

• k denotes the number of hash functions for cuckoo hashing [33].

• Given an element z on $\mathbb{F}_p$, z^{-1} denotes the inversion of z modulo p.

• $|\alpha|$ denotes the bit length of an integer α.

• $\langle \alpha, \beta \rangle$ denotes a key-value pair, where α is key and β is value.

• $\perp$ denotes decryption failure.

• Given an EC point P, $P[x]$ and $P[y]$ denote its x-coordinate and y-coordinate, respectively.

• Let $\mathbf{T}_1 = \{t_{1,i} = \langle i * G, i \rangle \mid i \in [0, 2^{\ell_1} - 1]\}$ be a hash table. We will define $\mathbf{T}'_1 = \{t_{1,i} = \langle i * G[x], i \rangle \mid i \in [1, 2^{\ell_1 - 1}]\}$ to replace $\mathbf{T}_1$ in section IV-B, and $t_{1,i}$ occupies only 64 bits after our memory overhead optimizations.

• Let $\mathbf{T}_2 = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [0, 2^{\ell_2} - 1]\}$ be a linear table. We will define $\mathbf{T}'_2 = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [1, 2^{\ell_2 - 1}]\}$ to replace $\mathbf{T}_2$ in section IV-C.

• $\mathbf{BT}_1$ and $\mathbf{BT}_2$ are binary trees in the form of arrays with subscripts starting from 0. $\mathbf{BT}_1[i]$ and $\mathbf{BT}_1[j]$ denote their $(i+1)$-th and $(j+1)$-th elements, respectively. $\mathbf{BT}_1[i,j]$ and $\mathbf{BT}_2[i,j]$ denote the elements of $\mathbf{BT}_1$ and $\mathbf{BT}_2$ corresponding to subscripts i to j, respectively.

B. EC-based Additively Homomorphic Encryption

We run $\mathsf{GroupGen}(1^\lambda)$ to obtain $\{\mathbb{E}_p, \mathbb{G}, G, N\}$. An EC-based AHE scheme contains the following algorithms.

• $\mathsf{KeyGen}(1^\lambda)$: KeyGen inputs a security parameter λ, then outputs the public key pk and private key sk.

• $\mathsf{Enc}(\mathrm{pk}, m)$: Enc inputs the public key pk and a message $m \in \{0,1\}^\ell$, then outputs a ciphertext $[\![m]\!]$.

• $\mathsf{Dec}(\mathrm{sk}, [\![m]\!])$: Dec inputs the private key sk and a ciphertext $[\![m]\!]$, then outputs m or $\perp$.

• $\mathsf{HomoAdd}(\mathrm{pk}, [\![m_1]\!], [\![m_2]\!])$: HomoAdd inputs two ciphertexts $[\![m_1]\!]$ and $[\![m_2]\!]$ encrypted by pk, then outputs a ciphertext $[\![m_1+m_2]\!]$.

• $\mathsf{ScalarMult}([\![m]\!], r)$: ScalarMult inputs a ciphertext $[\![m]\!]$ and a random scalar $r \in \mathbb{Z}_N^*$, then outputs $[\![r \cdot m]\!]$.

C. Exp-ElGamal

Exp-ElGamal is a simple variant of the original ElGamal [39], proposed by Cramer et al [12]. Recently, Exp-ElGamal has been accepted as the ISO standard [40]. It requires solving the ECDLP during the decryption process, and other EC-based AHE schemes [8]–[10], [13], [14] are similar. We run $\mathsf{GroupGen}(1^\lambda)$ to obtain $\{\mathbb{E}_p, \mathbb{G}, G, N\}$.

Exp-ElGamal scheme contains the following algorithms.

• $\mathsf{KeyGen}(1^\lambda)$: KeyGen inputs a security parameter λ, then outputs the public key $\mathrm{pk} = \mathrm{sk} * G$ and private key $\mathrm{sk} \in \mathbb{Z}_N^*$.

• $\mathsf{Enc}(\mathrm{pk}, m)$: Enc inputs the public key pk and a message $m \in \{0,1\}^\ell$, then outputs a ciphertext $[\![m]\!] = (c_1, c_2)$, where $c_1 = r * G, c_2 = m * G + r * \mathrm{pk}, r \in \mathbb{Z}_N^*$.

• $\mathsf{Dec}(\mathrm{sk}, [\![m]\!])$: Dec inputs the private key sk and a ciphertext $[\![m]\!] = (c_1, c_2)$, then computes $P_m = c_2 - \mathrm{sk} * c_1$. We get $m = \log_G P_m$ or $\perp$ over running an algorithm for solving the ECDLP.

• $\mathsf{HomoAdd}(\mathrm{pk}, [\![m_1]\!], [\![m_2]\!])$: HomoAdd inputs two ciphertexts $[\![m_1]\!] = (c_{1,1}, c_{1,2})$ and $[\![m_2]\!] = (c_{2,1}, c_{2,2})$ encrypted by pk, then outputs a ciphertext $[\![m_1+m_2]\!] = (c_{1,1}+c_{2,1}, c_{1,2}+c_{2,2})$.

• $\mathsf{ScalarMult}([\![m]\!], r)$: ScalarMult inputs a ciphertext $[\![m]\!] = (c_1, c_2)$ and a random scalar $r \in \mathbb{Z}_N^*$, then outputs $[\![r \cdot m]\!] = (r * c_1, r * c_2)$.

Remark. There is an elliptic curve $\mathbb{E}_p : y^2 = x^3 + ax + b \mod p$. The addition operation '+' on $\mathbb{E}_p$ is subject to the following calculation rules. Given two EC points $P_1 : (x_1, y_1)$, $P_2 : (x_2, y_2)$, the operation rule of $P_3 = P_1 + P_2$ is:

$$\begin{cases} P_3[x] = \tau^2 - (x_1 + x_2) \mod p \\ P_3[y] = \tau(x_1 - x_3) - y_1 \mod p, \end{cases} \tag{1}$$

where

$$\begin{cases} \tau = \frac{y_2 - y_1}{x_2 - x_1} \mod p \quad (P_1 = P_2) \\ \tau = \frac{3x_1^2 + a}{2y_1} \mod p, \quad (P_1 \neq P_2) \end{cases} \tag{2}$$

and a is the coefficient of $\mathbb{E}_p$. Furthermore, given an EC point $P_1 : (x_1, y_1)$, then $-P_1$ is $(x_1, p - y_1)$.

D. Baby-Step Giant-Step Algorithm (BSGS)

EC-based AHE schemes [8], [9], [12]–[14] encode the plaintext $m \in \{0,1\}^\ell$ as $m * G$ in order to obtain additive homomorphism. Therefore, we need to employ a dedicated algorithm to compute the ECDLP in an interval more efficiently. In this paper, we construct $\mathsf{FastECDLP}$ based on the BSGS algorithm [19] since it admits flexible time/space trade-offs and is amenable to parallelization. Let $\mathbf{T}_1 = \{t_{1,i} = \langle i * G, i\rangle \mid i \in [0, 2^{\ell_1} - 1]\}$ and $\mathbf{T}_2 = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [0, 2^{\ell_2} - 1]\}$, where $\ell_1 + \ell_2 = \ell$. The key-value table $\mathbf{T}_1$ and linear table $\mathbf{T}_2$ can be precomputed. The BSGS algorithm for EC-based AHE decryption is illustrated in Algorithm 1.

Algorithm 1 BSGS for EC-based AHE decryption

Input: $p, \ell_1, \ell_2, \ell = \ell_1 + \ell_2, \mathbf{T}_1 = \{t_{1,i} = \langle i * G, i\rangle \mid i \in [0, 2^{\ell_1} - 1]\}, \mathbf{T}_2 = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [0, 2^{\ell_2} - 1]\}, P_m = m * G$, where $m \in \{0,1\}^\ell$

Output: $m \in \{0,1\}^\ell$ or $\perp$

1: **for** $j = 0$ to $2^{\ell_2} - 1$ **do**
2: $\quad z_j = t_{2,j}[x] - P_m[x] \mod p$
3: $\quad$ **if** $z_j = 0$ **then**
4: $\quad\quad$ **return** $m = j \cdot 2^{\ell_1}$
5: $\quad$ **end if**
6: $\quad \eta = z_j^{-1} \mod p$
7: $\quad \tau = (p - t_{2,j}[y] - P_m[y]) \cdot \eta \mod p$
8: $\quad Q[x] = \tau^2 - (P_m[x] + t_{2,j}[x]) \mod p$
9: $\quad Q[y] = \tau(P_m[x] - Q[x]) - P_m[y] \mod p$
10: $\quad$ **if exist** $Q = t_{1,i}$ **then**
11: $\quad\quad$ **return** $m = j \cdot 2^{\ell_1} + i$
12: $\quad$ **end if**
13: **end for**
14: **return** $\perp$

Note that lines 3 to 5 can be sure that $P_m \notin \mathbf{T}_2$ such that $P_m \neq t_{2,j}$ holds. That is, if $P_m = t_{2,j}$, then m is directly obtained in line 4. Lines 2 to 9 are for calculating $Q = P_m - t_{2,j}$. In every loop, this process requires 6 modular additions, 3 modular multiplications, and one modular inversion. After benchmarking, we show the specific efficiency of various operations on $\mathbb{F}_p$ in Table III.

TABLE III
EFFICIENCY OF VARIOUS OPERATIONS ON $\mathbb{F}_p$, WHERE $|p| = 256$ FOR SECURITY (UNIT: NS).

operations	addition	multiplication	inversion
runtime	4	46	10346

According to Table III, the modular inversion (i.e., $\eta = z_j^{-1} \mod p$ in line 6) is very time-consuming. We need to compute at most 2^{ℓ_2} modular inversions for the decryption. They account for more than 98% of the computation overhead in Algorithm 1.

E. Cuckoo Hashing

Cuckoo hashing [33] is a kind of key-value data structure that uses k hash functions $h_i : \{0,1\}^* \rightarrow [1, \phi], i \in [1, k]$ to

determine the position of t elements, where ϕ is generally set to about $1.3t$ and $k = 3$. It is suitable for application scenarios with a large number of look-up operations and a small number of insertion operations. When a new element e is inserted into the cuckoo hashing, its position is $h_i(e)$, where $i \in [1, k]$. If there is already an element e' at that position, it will be moved to position $h_j(e')$, where $h_j(e') \neq h_i(e), j \in [1, k]$. The procedure is repeated until no more evictions are necessary or until a threshold number of relocations has been performed. If the insert operation cannot be completed, then the last element is put in a stash. When searching for an element e, it should be compared with the elements at position $h_i(e)$ and with the elements in the stash. According to Pinkas et al. [41], cuckoo hashing does not require an additional stash if $\phi = 1.3t$ and $k = 3$.

IV. FastECDLP: An Efficient Algorithm for Solving The Small Exponential ECDLP during the decryption of EC-based AHE schemes

In this section, we construct an efficient algorithm called FastECDLP based on the BSGS algorithm [19] for EC-based AHE decryption. FastECDLP can be applied to existing EC-based AHE schemes, such as Exp-ElGamal [12], BGN [13], and Twisted-ElGamal [8]. In these AHE schemes, the plaintext m needs to be recovered from an EC point $P_m = m * G$.

A. The Tree-based Montgomery's Trick

In Algorithm 1, the decryption requires at most 2^{ℓ_2} modular inversions, which is one of the crucial reasons for the poor decryption efficiency of EC-based AHE schemes. The Montgomery's trick [42] is an algorithm that allows for the efficient solution of multiple modular inversions. The core idea is to replace modular inversions with efficient modular multiplications. However, the computation process of [42] is serial and does not support parallel computation. Tree-based Montgomery's trick [32] is a variant that can support parallel computation. For the sake of completeness and readability of FastECDLP, we introduce the tree-based Montgomery's trick and show how to implement it. Since [32] is used to accelerate scalar multiplication rather than combining it with BSGS, the presentation of tree-based Montgomery's trick in this paper differs slightly from [32].

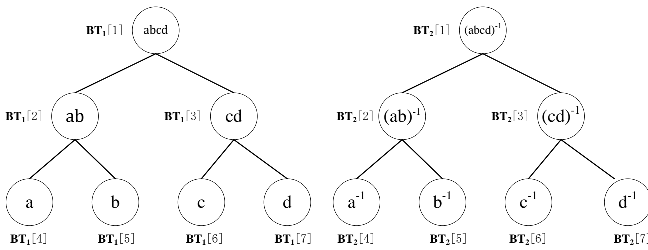

Fig. 1. Illustration for Algorithm 2 (BuildMulTree) and Algorithm 3 (BuildInvTree) with $\ell_2 = 2$.

For the clarity, we divide the tree-based Montgomery's trick into two parts. Specifically, we create two binary trees, $\mathbf{BT}_1$ and $\mathbf{BT}_2$, of height $\ell_2 + 1$ for each decryption, where all the modular inversions needed for the decryption are the leaf nodes of $\mathbf{BT}_2$, as shown in Figure 1. We show the readers how to implement it by pseudo-code form in Algorithm 2 and Algorithm 3.

Algorithm 2 Generation of $\mathbf{BT}_1$ (BuildMulTree)

Input: $p, \ell_2, \mathbf{Z} = \{z_j = t_{2,j}[x] - P_m[x] \mid j \in [0, 2^{\ell_2} - 1]\}, \mathbf{BT}_1 = \emptyset$
Output: $\mathbf{BT}_1$

1: **for** $j = 0$ to $2^{\ell_2} - 1$ **do**
2: $\quad \mathbf{BT}_1[j] = z_j$
3: **end for**
4: $f = 2^{\ell_2}$; $h = 2^{\ell_2}$; $i = 0$
5: **for** $l = 2$ to $\ell_2 + 1$ **do**
6: $\quad h = h/2$
7: $\quad$ **for** $j = 1$ to h **do**
8: $\quad\quad \mathbf{BT}_1[f] = \mathbf{BT}_1[i] \cdot \mathbf{BT}_1[i+1] \mod p$ (using asynchronous execution)
9: $\quad\quad f = f + 1$; $i = i + 2$
10: $\quad$ **end for**
11: $\quad$ Waiting for all the ℓ-th layer nodes to be generated, then starting the next outside loop.
12: **end for**
13: **return** $\mathbf{BT}_1$

$\mathbf{BT}_1$ and $\mathbf{BT}_2$ are complete binary trees in the form of arrays that store elements in a level order. The construction rule of $\mathbf{BT}_1$ is that the parent node is obtained by modular multiplication of two child nodes. Therefore, the root node of $\mathbf{BT}_1$ is the modular multiplications of all leaf nodes. We let $\mathbf{Z} = \{z_j = t_{2,j}[x] - P_m[x] \mid j \in [0, 2^{\ell_2} - 1]\}$, where $t_{2,j} \in \mathbf{T}_2$. We can generate $\mathbf{BT}_1$ using the elements in $\mathbf{Z}$ as the leaf nodes of $\mathbf{BT}_1$. Note that $z_j \neq 0$ because if $z_j = 0$, m is returned in line 4 of Algorithm 1. The generation process of $\mathbf{BT}_1$ is shown in Algorithm 2, which requires $2^{\ell_2} - 1$ modular multiplications.

Algorithm 3 Generation of binary tree $\mathbf{BT}_2$ (BuildInvTree)

Input: $p, \ell_2, \mathbf{BT}_1, \mathbf{BT}_2 = \emptyset$
Output: $\mathbf{BT}_2$

1: $f = 2^{\ell_2+1} - 2$; $k = 1$
2: $\mathbf{BT}_2[f] = (\mathbf{BT}_1[f])^{-1} \mod p$
3: **for** $i = \ell_2$; $i > 0$; $i = i - 1$ **do**
4: $\quad h = f - 2 \cdot k$
5: $\quad$ **for** $j = 0$ to $2 \cdot k - 1$ **do**
6: $\quad\quad v = j + h$; $w = v \oplus 1$
7: $\quad\quad \mathbf{BT}_2[v] = \mathbf{BT}_1[w] \cdot \mathbf{BT}_2[f + (j/2)] \mod p$ (using asynchronous execution)
8: $\quad$ **end for**
9: $\quad f = h$; $k = 2 \cdot k$
10: $\quad$ Waiting for all the i-th layer nodes to be generated, then starting the next loop.
11: **end for**
12: **return** $\mathbf{BT}_2$

After generating $\mathbf{BT}_1$, we can use $\mathbf{BT}_1$ to generate $\mathbf{BT}_2$. The generation process is illustrated in Algorithm 3, which

includes $2^{\ell_2+1}-2$ modular multiplications and one modular inversion. Contrary to $\mathbf{BT}_1$, which is generated bottom-up, $\mathbf{BT}_2$ is generated top-down, and its root node is the inverse element of the root node of $\mathbf{BT}_1$. The rule for the generation of $\mathbf{BT}_2$ is that the node value is equal to the parent node multiplied by the brother node position of $\mathbf{BT}_1$. The XOR operation '$\oplus$' is a simple trick to quickly find the position of the brother node in a binary tree.

$\mathbf{BT}_1$ and $\mathbf{BT}_2$ require layer-by-layer generation, i.e., the creation of nodes on the same layer can be parallelized. Meanwhile, $\mathbf{BT}_1$ and $\mathbf{BT}_2$ can share an array in the specific implementation. Therefore, a certain amount of memory can be saved. Moreover, after $\mathbf{BT}_2$ is generated, the remaining memory, i.e., non-leaf nodes $\mathbf{BT}_2[2^{\ell_2}, 2^{\ell_2+1}-2]$, can be freed to optimize the memory overhead.

We now merge Algorithm 2 (BuildMulTree) and Algorithm 3 (BuildInvTree) to show the complete tree-based Montgomery's trick. That is, given $\mathbf{Z} = \{z_j = t_{2,j}[x] - P_m[x] \mid j \in [0, 2^{\ell_2}-1]\}$, we can efficiently compute all the inversions $\mathbf{Z}' = \{z'_j = z_j^{-1} \mid j \in [0, 2^{\ell_2}-1]\}$.

Algorithm 4 Tree-based Montgomery's trick (TreeMon)

Input: $p, \ell_2, \mathbf{Z} = \{z_j = t_{2,j}[x] - P_m[x] \mid j \in [0, 2^{\ell_2}-1]\}$
Output: $\mathbf{Z}' = \{z'_j = z_j^{-1}\}$
1: $\mathbf{BT}_1 \leftarrow \mathsf{BuildMulTree}(p, \ell_2, \mathbf{Z})$
2: $\mathbf{BT}_2 \leftarrow \mathsf{BuildInvTree}(p, \ell_2, \mathbf{BT}_1)$
3: **free** $\mathbf{BT}_1$
4: **free** $\mathbf{BT}_2[2^{\ell_2}, 2^{\ell_2+1}-2]$
5: **return** $\mathbf{Z}' = \mathbf{BT}_2[0, 2^{\ell_2}-1]$

Due to Algorithm 4 (TreeMon), the decryption of EC-based AHE schemes no longer do so many modular inversions (line 6) in Algorithm 1. Therefore, the decryption efficiency is considerably improved. In other words, we have transformed 2^{ℓ_2} modular inversions into one modular inversion and $3 \cdot 2^{\ell_2} - 3$ modular multiplications using TreeMon.

Remark. According to Table III, executing one modular multiplication and one modular inversion take 46 ns and 10346 ns, respectively. Consequently, we can enhance the efficiency of this step by as much as at least 75 times. In addition, BSGS with TreeMon is the basic skeleton of our FastECDLP. However, simply applying TreeMon to BSGS for solving ECDLP during the decryption is still insufficient since it still involves numerous table look-up operations, modular multiplications, and a huge memory overhead caused by $\mathbf{T}_1$ and $\mathbf{T}_2$. Meanwhile, if we use a classic hash table, such as hashmap, to store $\mathbf{T}_1$, this is still a huge memory overhead since hashmap will cause about 8 times memory redundancy in practical implementation. Therefore, we need to do further optimizations in following subsections regarding computation and memory overhead to ensure efficient decryption.

B. Memory Overhead Optimizations

In this subsection, we show how to optimize FastECDLP's memory overhead. The majority of FastECDLP's memory overhead consists of precomputed $\mathbf{T}_1 = \{\langle i * G, i\rangle \mid i \in [0, 2^{\ell_1}-1]\}$ and $\mathbf{T}_2 = \{(j \cdot 2^{\ell_1}) * G \mid j \in [0, 2^{\ell_2}-1]\}$ since $\mathbf{T}_1$ and $\mathbf{T}_2$ should be in memory during the decryption process of EC-based AHE schemes. The memory overhead caused by $\mathbf{T}_2$ is $2^{\ell_2} \cdot (|p| + 8)$ bits (for security, $|p| = 256$). That is, only one byte is needed to denote the positive or negative of the y-coordinate. Meanwhile, there is no memory redundancy for the linear table in practical implementation. Therefore, the memory overhead caused by $\mathbf{T}_2$ is small. For example, when ℓ_2 takes a maximum of 23 in FastECDLP, $\mathbf{T}_2$ takes up only 0.25 GB[2] of memory. What needs to be reduced is the memory overhead associated with $\mathbf{T}_1$. We use the following three methods to optimize the memory overhead caused by $\mathbf{T}_1$.

1) Only store the x-coordinates in $\mathbf{T}_1$: In line 10 of Algorithm 1, we should determine whether there is an element $t_{1,i} \in \mathbf{T}_1$ such that $Q = t_{1,i}$ holds, where $i \in [0, 2^{\ell_1}-1]$, $Q = P_m - t_{2,j}$, and $t_{2,j} \in \mathbf{T}_2$. Thus, if we want to determine whether there is an EC point Q in $\mathbf{T}_1$, it is sufficient to store $\{(i * G)[x] \mid i \in [0, 2^{\ell_1}-1]\}$. Furthermore, since $(i * G)[x] = (-i * G)[x]$, we can use $(i * G)[x]$ to correspond to $\{i, -i\}$. That is, $\mathbf{T}_1$ can represent 2^{ℓ_1+1} baby steps. We only need 2^{ℓ_1} baby steps to cover the giant step. Therefore, we can define $\mathbf{T}'_1 = \{t_{1,i} = \langle i * G[x], i\rangle \mid i \in [1, 2^{\ell_1-1}]\}$ to replace $\mathbf{T}_1$. The plaintext space is still $\{0,1\}^{\ell}$, reducing the memory overhead by 2 times.

2) Truncating the x-coordinates while ensuring no collisions occur: Storing the whole x-coordinates, i.e., $\{(i * G)[x] \mid i \in [1, 2^{\ell_1-1}]\}$, is also unnecessary since a part of them is enough to denote keys for a hash table without collision. For example, storing the first θ bits of $(i * G)[x]$ is sufficient as long as no collision occurs. In our experimental part, ℓ_1 takes a maximum of 31. According to the birthday attack[3], the probability of a collision is about

$$1 - e^{\frac{-3 \cdot 2^{\ell_1-1}(3 \cdot 2^{\ell_1-1}-1)}{2\theta}}. \tag{3}$$

If $\theta = 64$ and $\ell_1 = 31$, then the Equation (3) is approximately equal to 0.24. We have tested that there is no collision occurs at 2^{ℓ_1-1} EC points when $\ell_1 = 31$. Therefore, we can save about 4 times the memory overhead by storing only 64-bit x-coordinates for $\mathbf{T}'_1$. Note that we actually do not care what the probability of a collision is, because all we need to do is determine the minimum value of θ on the premise that no collision occurs in practice when $\ell_1 = 31$. Therefore, if $\ell_1 < 31$, then θ can be set to smaller than 64, further reducing the memory overhead.

3) Using cuckoo hashing [33] instead of a classic hash table: The typical hash table (i.e., hashmap) causes a lot of memory redundancy, leading to memory waste. According to our benchmarking, hashmap leads to 8 times more memory redundancy. Cuckoo hashing only require $1.3 \cdot 2^{\ell_1-1}$ bins for $\mathbf{T}'_1 = \{t_{1,i} = \langle i * G[x], i\rangle \mid i \in [1, 2^{\ell_1-1}]\}$. Note that cuckoo hashing is essentially two linear tables of size $1.3 \cdot 2^{\ell_1-1}$, representing keys and values, respectively. Therefore, there is no additional redundancy in practical implementation. Cuckoo hashing saves about 5 times more memory than hashmap. If

[2]After $\mathbf{T}_2$ is optimized to $\mathbf{T}'_2$, it is actually only about 0.12 GB.

[3]We use cuckoo hashing to store $\mathbf{T}'_1$ rather than the hashmap. Therefore, there are actually at most $3 \cdot 2^{\ell_1-1}$ hash functions when $k = 3$.

storing $i \in [1, 2^{\ell_1-1}]$ with the uint32 type, then $\mathbf{T}_1'$ takes about $1.3 \cdot (64+32) \cdot 2^{\ell_1-1}$ bits of memory. We will continue with the optimization to $1.3 \cdot 64 \cdot 2^{\ell_1-1}$ bits, which will be presented together with the performance optimization of cuckoo hashing.

C. Further Computation Overhead Optimizations

In this subsection, we describe how to optimize FastECDLP in terms of computation overhead. This part further improves the efficiency of FastECDLP and ensures efficient decryption. We use the following three methods to optimize the computation overhead.

1) Computing $(t_{2,j} - P_m)[x]$ using $(P_m - t_{2,j})[x]$: After our previous optimizations, the essence of solving ECDLP is to find $i \in [1, 2^{\ell_1-1}]$ and $j \in [1, 2^{\ell_2}]$ such that

$$(P_m - t_{2,j})[x] = i * G[x] \in \mathbf{T}_1' \quad (4)$$

holds, where $P_m = m * G$ and $m = j \cdot 2^{\ell_1} \pm i$. Since $\mathbf{T}_1'$ only stores the x-coordinates, line 9 of Algorithm 1 (calculating $(P_m - t_{2,j})[y]$) can be ignored, saving one modular multiplication in every loop. We know that $(i * G)[x] = (-i * G)[x]$ in $\mathbf{T}_1'$. Therefore, we should transform Equation (4) into

$$(t_{2,j} - P_m)[x] = (-i * G)[x] \in \mathbf{T}_1'. \quad (5)$$

$(P_m - t_{2,j})[x]$ and $(t_{2,j} - P_m)[x]$ need to compute $(-t_{2,j}[x] - P_m[x])^{-1}$ and $(-P_m[x] - t_{2,j}[x])^{-1}$, respectively. Since $P[x] = -P[x]$ holds for any given EC point P, we can see that $(P_m - t_{2,j})[x]$ and $(t_{2,j} - P_m)[x]$ share the same modular inversions. It means that when ℓ is constant, we only require half of the modular inversions. That is, we can transform the plaintext space from $\{0,1\}^\ell$ to $[-2^{\ell-1}, 2^{\ell-1}-1]$[4], thereby reducing 2^{ℓ_2} modular inversions to 2^{ℓ_2-1} modular inversions in Algorithm 4. Therefore, $\mathbf{T}_2$ can store half of the elements. We can define $\mathbf{T}_2' = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [1, 2^{\ell_2-1}]\}$ to replace $\mathbf{T}_2$. The way to recover the plaintext m becomes to find $i \in [1, 2^{\ell_1-1}]$ and $j \in [1, 2^{\ell_2-1}]$ such that Equation (4) or Equation (5) holds. Then, we judge $m \in \{j \cdot 2^{\ell_1} + i, j \cdot 2^{\ell_1} - i, -j \cdot 2^{\ell_1} + i, -j \cdot 2^{\ell_1} - i\}$ such that $P_m = m * G$ holds.

We replace $\mathbf{T}_1$ and $\mathbf{T}_2$ of sizes 2^{ℓ_1} and 2^{ℓ_2} with $\mathbf{T}_1'$ and $\mathbf{T}_2'$ of sizes 2^{ℓ_1-1} and 2^{ℓ_2-1}, respectively. Therefore, we can solve the ECDLP of scale ℓ with scale $\ell - 2$. The computation and memory overhead can be balanced by adjusting ℓ_1 and ℓ_2. Thus, if ℓ is constant, then the computation overhead can be reduced by 4 times. If the computation overhead is constant, then the memory overhead can be reduced by 4 times.

2) Using the coordinate values directly as hash values of cuckoo hashing: In line 10 of Algorithm 1, there are a large number of hash table look-up operations, which also means a large computation overhead for the cuckoo hashing. To further improve the efficiency, we should boost the query speed of the cuckoo hashing. The cuckoo hashing requires computing k small hash values, typically only 32 bits, when performing a look-up operation. We can split the coordinates of points into multiple hash values to avoid a lot of hash functions during decryption. In addition, we also discuss how to optimize memory overhead for $\mathbf{T}_1'$ using the truncation of coordinates. For example, Figure 2 shows that for $k = 3$, we can treat $(i * G)[x]$ as the 3 hash values of cuckoo hashing. The hash values point to the indexes of the bins, and they do not need to be stored. The 32-bit values connected by the hash values in Figure 2 are stored in the bins as keys. Thus, only $1.3 \cdot (32+32) \cdot 2^{\ell_1-1}$ bits are needed to store $\mathbf{T}_1'$ since the values $i \in [1, 2^{\ell_1-1}]$ use the uint32 type.

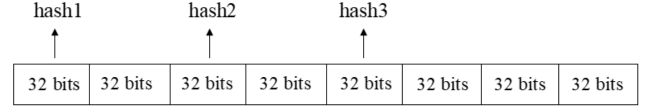

Fig. 2. The x-coordinate is considered the hash values and keys of the cuckoo hashing when $k = 3$. ($|p| = 256$)

The hash values and the keys together form 64 bits to ensure finding the correct i from $\mathbf{T}_1'$. According to Equation (3), the probability of a collision is about 24% when $\ell_1 = 31$ and $\theta = 64$. It should be emphasized that we only need to ensure that 2^{ℓ_1-1} key-value pairs can be successfully inserted into the bins of cuckoo hashing. We show the probability just to prove that our method is worth trying. Since we only need a suitable method to split the x-coordinates, and as long as we successfully obtain $\mathbf{T}_1'$ once, $\mathbf{T}_1'$ can be used for all the decryptions in this setting (the curve and ℓ_1).

The above optimizations can speed up each look-up operation by about 5 to 10 ns. In the experimental part, we will show the specific experimental results of our cuckoo hashing. Since the decryption of EC-based AHE schemes requires at most $k \cdot 2^{\ell_2-1}$ hash functions, even a single-query nanosecond level improvement is meaningful concerning many large-scale practical applications [8], [25], [29].

3) The implementation of $\mathbb{F}_p$: The entire process of recovering the plaintext m is operations on the finite field $\mathbb{F}_p$. At the same time, we can see that Algorithm 4 brings a host of additional modular multiplications. Therefore, we have to optimize the implementation of $\mathbb{F}_p$. The most straightforward representation would be using an array of 4 uint64s ($64 \cdot 4 = 256$). However, there is no space left for overflow when intermediate calculations are performed between two arrays, resulting in expensive carry propagation. The representation in our paper is ten uint32s with base 2^{26} ($26 \cdot 10 = 260$, so the last word needs only 22 bits), leaving the desired 64 bits ($32 \cdot 10 = 320, 320 - 256 = 64$) for overflow. This representation was inspired by the implementation of the secp256k1 curve in Bitcoin [43]. It can speed up the whole process of solving the small exponential ECDLP by about 2 to 3 times.

D. Exp-ElGamal Decryption using Our FastECDLP

We take Exp-ElGamal [12] as the example to describe how FastECDLP is embedded in an EC-based AHE scheme. Note that FastECDLP is generic to any EC-based AHE scheme and that the method embedded in other schemes [8], [9], [13], [14] is similar to that of Exp-ElGamal, which is not repeated in this paper.

Let $\mathsf{Search}(\mathbf{T}_1', \mathbb{F}_p) \to \{i, \mathrm{ok}\}$ be the input of a cuckoo hashing $\mathbf{T}_1'$ and $P[x] \in \mathbb{F}_p$. If $P[x]$ is in $\mathbf{T}_1'$, then it

[4] Note that the size of the plaintext space has not changed. Subtracting $2^{\ell-1}$ from the upper and lower bounds of $\{0,1\}^\ell$ gives $[-2^{\ell-1}, 2^{\ell-1}-1]$.

returns $\mathrm{ok} = \mathrm{true}$ and the index i; otherwise, $\mathrm{ok} = \mathrm{false}$ and $i = \emptyset$. Following the optimizations in this section, we show the complete FastECDLP in Algorithm 5, which can ensure efficient decryption EC-based AHE schemes when the plaintext length ℓ is as long as possible. The computational complexity and required memory space of FastECDLP are $\mathcal{O}(2^{\ell_2-1})$ and $1.3 \cdot 64 \cdot 2^{\ell_1-1} + 264 \cdot 2^{\ell_2-1}$ bits, respectively.

Algorithm 5 Solving ECDLP fastly (FastECDLP)

Input: p, ℓ_1, ℓ_2, ℓ, $\mathbf{T}'_1 = \{t_{1,i} = \langle i * G[x], i\rangle \mid i \in [1, 2^{\ell_1-1}]\}$ with the memory optimizations, $\mathbf{T}'_2 = \{t_{2,j} = (j \cdot 2^{\ell_1}) * G \mid j \in [1, 2^{\ell_2-1}]\}, P_m = m * G$, where $m \in [-2^{\ell-1}, 2^{\ell-1} - 1]$
Output: m or $\perp$

1: $\mathbf{Z} \leftarrow \emptyset$
2: // Determine whether P_m is in $\mathbf{T}'_1$.
3: **if** $\{i, \mathrm{ok}\} \leftarrow \mathsf{Search}(\mathbf{T'_1}, P_m)$; ok **then**
4: $m_1 = i$; $m_2 = -i$
5: **goto** line 33
6: **end if**
7: // Determine whether P_m is in $\mathbf{T}'_2$ and generate $\mathbf{Z} = \{z_j\}$.
8: **for** $j \in [1, 2^{\ell_2-1}], t_{2,j} \in \mathbf{T}'_2$ **do**
9: $z_j = t_{2,j}[x] - P_m[x]$
10: **if** $z_j = 0$ **then**
11: $m_1 = j \cdot 2^{\ell_1}$; $m_2 = -j \cdot 2^{\ell_1}$
12: **goto** line 33
13: **end if**
14: $\mathbf{Z} = \mathbf{Z} \cup \{z_j\}$
15: **end for**
16: // The modular inversions are calculated by Algorithm 4.
17: $\mathbf{Z}' = \{z'_j = z_j^{-1} \mid j \in [1, 2^{\ell_2-1}]\} \leftarrow \mathsf{TreeMon}(p, \ell_2 - 1, \mathbf{Z})$
18: **for** $j \in [1, 2^{\ell_2-1}], t_{2,j} \in \mathbf{T}'_2$ **do**
19: $\varphi = P_m[x] + t_{2,j}[x] \bmod p$
20: // Determine whether Equation (4) holds.
21: $Q[x] = ((p - t_{2,j}[y] - P_m[y]) \cdot z'_j)^2 - \varphi \bmod p$
22: **if** $\{i, \mathrm{ok}\} \leftarrow \mathsf{Search}(\mathbf{T'_1}, Q[x])$; ok **then**
23: $m_1 = j \cdot 2^{\ell_1} + i$; $m_2 = j \cdot 2^{\ell_1} - i$
24: **break**
25: **end if**
26: // Determine whether Equation (5) holds.
27: $Q[x] = ((t_{2,j}[y] - P_m[y]) \cdot z'_j)^2 - \varphi \bmod p$
28: **if** $\{i, \mathrm{ok}\} \leftarrow \mathsf{Search}(\mathbf{T'_1}, Q[x])$; ok **then**
29: $m_1 = -j \cdot 2^{\ell_1} - i$; $m_2 = -j \cdot 2^{\ell_1} + i$
30: **break**
31: **end if**
32: **end for**
33: **if** $m_1 * G = P_m$ **then**
34: **return** $m = m_1$
35: **else if** $m_2 * G = P_m$ **then**
36: **return** $m = m_2$
37: **end if**
38: **return** $\perp$

The encryption and decryption process of Exp-ElGamal combined with FastECDLP is as follows.

• $\mathsf{Enc}(\mathrm{pk}, m)$: Enc inputs the public key pk and a message $m \in [-2^{\ell-1}, 2^{\ell-1} - 1]$, then outputs a ciphertext $[\![m]\!] = (c_1, c_2)$, where $c_1 = r * G, c_2 = m * G + r * \mathrm{pk}$.

• $\mathsf{Dec}(\mathrm{sk}, [\![m]\!])$: Dec inputs the private key sk and a ciphertext $[\![m]\!] = (c_1, c_2)$, then computes $\mathrm{sk}*c_1$. If $\mathrm{sk}*c_1 = c_2$ holds, then return $m = 0$; otherwise compute $P_m = c_2 - \mathrm{sk} * c_1$. We run FastECDLP to get $m = \log_G P_m$ or $\perp$.

Remark. FastECDLP can actually recover $m \in [-2^{\ell-1} - 2^{\ell_1-1}, 2^{\ell-1} + 2^{\ell_1-1}]$ from P_m. In order to keep the size of the plaintext space unchanged, we still set m to $[-2^{\ell-1}, 2^{\ell-1} - 1]$ in Algorithm 5. In other words, FastECDLP can handle the size of $2^\ell + 2^{\ell_1}$, which can further prevent overflow in practical applications.

V. Experiments

This section introduces the experimental details and presents the experimental results, demonstrating that FastECDLP can ensure efficient decryption and maximizing $\ell = \ell_1 + \ell_2$.

A. Experiment Setup

Experimental environment is Ubuntu 20.04 with Intel(R) Xeon(R) Gold 5218 (2.30 GHz), 32 cores, and 98 GB RAM. The security level of AHE is 128-bit, that is, the RSA modulus n is 3072 bits, and p is 256 bits. The plaintext length ℓ is set from 32 to 54. ℓ_1 is set from 21 to 31, and ℓ_2 is set from 11 to 23. This ensures that decryption can be completed within 1 second and the memory overhead is within 12 GB. EC-based AHE schemes are implemented in the go language with the secp256k1 curve [44]. For cuckoo hashing, k is set to 3, and the number of bins is set to $1.3 \cdot 2^{\ell_1-1}$.

B. Experimental results

1) Benchmarks for cuckoo hashing: We benchmark our customized cuckoo hashing, which is an essential fundamental component for FastECDLP. We use the hashmap and our cuckoo hashing to store $\mathbf{T}'_1 = \{t_{1,i} = \langle i * G[x], i\rangle \mid i \in [1, 2^{\ell_1-1}]\}$ by varying ℓ_1. As can be seen in Figure 3, the hashmap and our cuckoo hashing need about 60 GB and 11 GB of memory when $\ell_1 = 31$, respectively. It is worth noting that [24] also compresses $\{\langle i * G[x], i\rangle\}$ to 64-bit values with another point compression method, but the data structure is the hashmap. Therefore, FastECDLP can save about 5 times as much memory as [24] with our cuckoo hashing. Moreover, the point compression method in [24] requires a small amount of computation, while we do not, since we directly truncate the coordinates as hash values and keys for cuckoo hashing.

We then give the efficiency of the look-up operation in our cuckoo hashing and compare it to the original cuckoo hashing and the hashmap. As shown in Figure 4, our cuckoo hashing performs a look-up operation 5 to 10 ns faster than hashmap and original cuckoo hashing. Note that we are not concerned here with the efficiency of insertions and deletions since they are never used after $\mathbf{T}'_1$ is generated.

2) Exp-ElGamal combined with FastECDLP*:* We combine FastECDLP with Exp-ElGamal [12] to show experimental results with the plaintext length ranging from 32 to 54 bits. Due to limited space, we choose some representative results in Table IV.

We can see from Table IV that FastECDLP combines well with concurrent programming and performs at its best

TABLE IV
DECRYPTION EFFICIENCY AND MEMORY ABOUT EXP-ELGAMAL COMBINED WITH FastECDLP (UNIT: MS, AVERAGE OF 1000 RUNS, $\ell = \ell_1 + \ell_2$). T DENOTES THE NUMBER OF THREADS. NOTE THAT $\ell_1 - 1$ CORRESPONDS TO THE SIZE OF $\mathbf{T}'_1$, AND $\ell_2 - 1$ CORRESPONDS TO THE COMPUTATION OVERHEAD AND THE SIZE OF $\mathbf{T}'_2$.

Plaintext length ℓ	ℓ_2	ℓ_1	FastECDLP (T = 1)	FastECDLP (T = 2)	FastECDLP (T = 4)	FastECDLP (T = 16)	Memory (GB)
32	11	21	1.02	0.57	0.41	0.35	0.010
33	11	22	1.01	0.58	0.42	0.36	0.021
34	11	23	1.02	0.55	0.43	0.35	0.043
35	11	24	0.99	0.52	0.44	0.36	0.086
36	11	25	1.04	0.56	0.45	0.37	0.173
37	11	26	0.87	0.57	0.46	0.39	0.345
38	11	27	0.89	0.48	0.37	0.31	0.690
39	11	28	0.83	0.57	0.36	0.32	1.380
40	11	29	0.78	0.51	0.35	0.28	2.761
41	11	30	1.01	0.58	0.45	0.36	5.522
42	11	31	1.02	0.56	0.41	0.35	11.044
43	12	31	1.83	1.19	0.83	0.54	11.044
44	13	31	3.68	2.37	1.62	1.14	11.044
45	14	31	8.01	4.48	3.20	2.41	11.045
46	15	31	14.80	8.81	6.44	4.44	11.045
47	16	31	27.48	16.78	11.62	7.93	11.046
48	17	31	58.50	35.55	23.91	17.08	11.047
49	18	31	122.42	72.45	50.13	34.14	11.049
50	19	31	237.36	141.80	95.33	63.93	11.052
51	20	31	502.59	292.44	198.56	127.15	11.060
52	21	31	922.24	547.70	371.01	248.13	11.076
53	22	31	1892.31	1103.24	764.29	501.43	11.107
54	23	31	4164.85	2436.23	1587.77	989.78	11.170

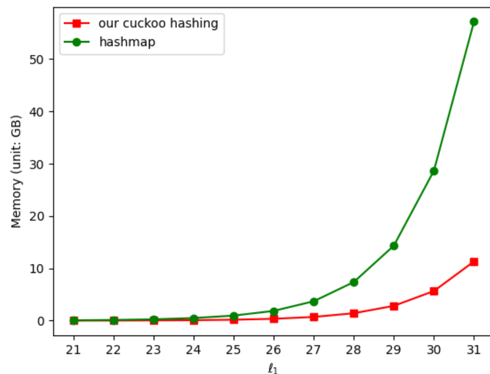

Fig. 3. Memory comparison between hashmap and our cuckoo hashing by varying ℓ_1 (unit: GB).

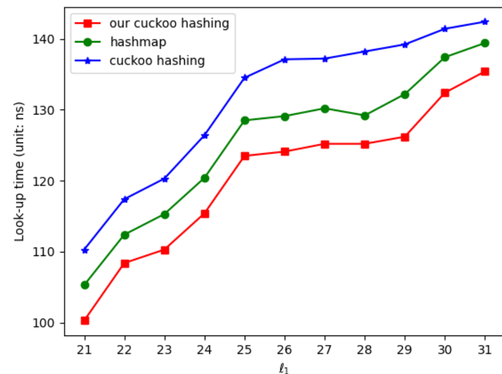

Fig. 4. Look-up running time comparison with hashmap, cuckoo hashing, and our cuckoo hashing by varying ℓ_1 (unit: ns).

performance in 16 thread. For example, when $\ell_1 = 21$ and $\ell_2 = 11$ (i.e., $\ell = 32$), FastECDLP requires 1.02 ms and 0.35 ms with a single thread and 16 threads, respectively. When $\ell_1 = 31$, $\ell_2 = 23$ (i.e., $\ell = 54$), FastECDLP takes about 1 second to complete a decryption with 16 threads. Compared to the decryption efficiency of RSA-based AHE schemes, this seems relatively low, but it is acceptable. First, few applications require encryption of a 54-bit number for additive homomorphic operations. Furthermore, although the decryption efficiency is relatively low, the advantages of EC-based AHE schemes are significant in other aspects, such as the encryption efficiency, ciphertext length, and homomorphic operations.

The computational complexity of FastECDLP is $\mathcal{O}(2^{\ell_2 - 1})$. Therefore, the efficiency of FastECDLP depends mainly on ℓ_2. We show the decryption efficiency of ℓ_2 from 11 to 23 under 16 threads in Figure 5. It shows that the decryption time almost doubles when ℓ_2 is increased by one and that FastECDLP is very efficient when $\ell_2 \leq 18$.

3) FastECDLP is generic for EC-based AHE schemes: FastECDLP is generic to any EC-based AHE scheme since they all need to solve the small exponential ECDLP during decryption. We show the decryption efficiency of mainstream EC-based AHE schemes after applying our FastECDLP when $\ell_1 = 29$ and $\ell_2 = 11$ with a single thread in Table V. That is, the plaintext space is $[-2^{39}, 2^{39}]$, and the plaintext length $\ell = 40$. The reason for choosing 40 bits is that this is an appropriate length for some applications, e.g., vertical federated learning and online auction systems, in this paper.

Any EC-based AHE scheme combined with FastECDLP

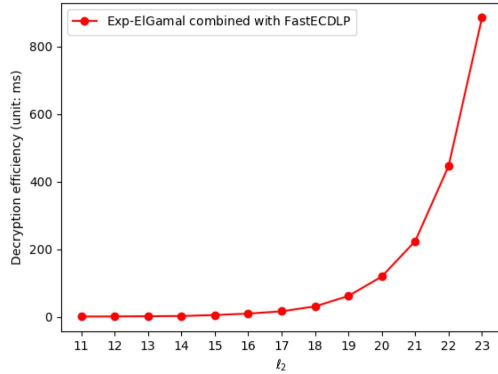

Fig. 5. Decryption efficiency under 16 threads by varying ℓ_2 (unit: ms).

TABLE V
DECRYPTION EFFICIENCY COMPARISON USING THE BSGS ALGORITHM [19] AND FastECDLP WITH A SINGLE THREAD ($\ell = 40, \ell_1 = 29, \ell_2 = 11$, AVERAGE OF 1000 RUNS).

Schemes	BSGS [19] (ms)	FastECDLP (ms)	vs. BSGS
Exp-ElGamal [12]	10.65	0.33	↑ 32×
BGN [13]	11.14	0.35	↑ 32×
Twisted-ElGamal [8]	11.88	0.34	↑ 35×

can perform decryption within about 0.35 ms with a single thread if $\ell = 40$, which is more than 30 times better than the scheme combined with BSGS [19], i.e., the original scheme. Therefore, FastECDLP can increase the decryption speed by more than 30 times for any EC-based AHE scheme when $\ell = 40$. Moreover, the decryption efficiency of Exp-ElGamal [12] combined with FastECDLP surpasses that of all RSA-based AHE schemes when $\ell = 40$. For example, according to Table II, OU [15] and Paillier [3] require about 1.3 ms and 9.5 ms to execute the decryption, respectively. Note that the decryption efficiency of RSA-based AHE schemes is hardly affected by ℓ. The decryption efficiency of any EC-based AHE scheme combined with FastECDLP is about 4 and 30 times faster than OU and Paillier, respectively.

VI. APPLICATIONS

In this section, using Exp-ElGamal [12] as the example, we show three specific applications of EC-based AHE schemes with the blessing of FastECDLP. In addition, we also report some specific experimental results of these applications. These show that EC-based AHE schemes have advantages in computation and communication overhead over RSA-based AHE schemes after applying FastECDLP. This further illustrates the great practical significance of FastECDLP in AHE scenarios.

A. Application to Federated Learning

Federated Learning (FL) [45] enables participants to perform joint model training without revealing source data. We take Vertical Federated Learning (VFL) as an example, which describes the case where data is vertically partitioned by features. Define two parties $\mathcal{A}$ and $\mathcal{B}$, both of whom wish to train a machine learning model by consolidating their respective data $\mathbf{D}_a$ and $\mathbf{D}_b$. $\mathcal{A}$ and $\mathcal{B}$ have the same batch of samples but different features, i.e., $\mathbf{D}_a$ and $\mathbf{D}_b$ vertically partitioned. The VFL system obtains $\mathbf{D} = \mathbf{D}_a \cap \mathbf{D}_b$ (similar to a "join" operation in databases) to train a model $\mathbf{M}$, in which process $\mathcal{A}$ or $\mathcal{B}$ does not expose its data to each other. Most existing VFL frameworks [7], [25], [34], [35] are constructed based on Paillier [3]. However, Paillier needs to choose a 3072-bit RSA modulus at 128-bit security. Therefore, the AHE operations of Paillier are inefficient, resulting in low efficiency of the existing VFL frameworks.

We can apply Exp-ElGamal [12] combined with FastECDLP to existing Paillier-based VFL frameworks to speed up the speed of model training. Meanwhile, since the short ciphertext length of Exp-ElGamal, communication overhead also can be reduced. We choose four well-known Paillier-based VFL frameworks, including two federated logistic regression frameworks (HeteroLR [25] and VFLwC [34]) and two federated gradient boosting decision tree frameworks (SecureBoost [35] and VF^{2}Boost [7]). The distributed architecture of VFL is simulated using Docker. We evaluate the performance using three datasets: MNIST [46], Ionosphere [47], and Breast Cancer [48], which are classic datasets and are suitable for VFL model training. For an FL system, 40-bit is a sufficient plaintext length since the model parameters that need to be encrypted, such as gradients or labels, are very small. For example, the gradients are only about 32 bits [6]. However, the result of scalar multiplication can easily overflow the plaintext space when $\ell = 32$. Therefore, to be on the safe side, we set the plaintext length parameters ℓ_1 and ℓ_2 to 29 and 11, respectively. That is, the plaintext length is $\ell = 40$. According to Table IV, the memory overhead is only 2.76 GB. It shows that FastECDLP does not overload VFL participants. Even a lightweight virtual container can run FastECDLP. Note that the accuracy of the jointly trained models is hardly affected by $\ell = 40$.

We report the concrete results under a single thread in Table VI on the training time and communication overhead for every VFL framework. Table VI shows that Exp-ElGamal combined with FastECDLP can improve can speed up model training of existing Paillier-based VFL frameworks by at most about 4 to 14 times. The interactive data of the VFL frameworks is predominantly the ciphertexts. Therefore, the communication largely depends on the ciphertext length. The ciphertext length of Paillier is 6144 bits, while that of Exp-ElGamal is only 528 bits, according to Table II. Therefore, the overall communication overhead can be reduced by a factor of 10 to 15.

In summary, our work has solid practical significance in the VFL scenarios.

B. Application to Blockchain

Blockchain is a sort of tamper-proof digital ledger of transactions in chronological order maintained by distributed consensus nodes (called miners). It is derived from a decentralized peer-to-peer digital currency system called Bitcoin [49]. Digital currency can be regarded as the most successful appli-

TABLE VI
RESULTS OF WHOLE TRAINING TIME AND COMMUNICATION PER ITERATION. COMMUNICATION OF EACH ITERATION IS OBTAINED BY PORT LISTENING.

Frameworks	Datasets	Training time (min)			Communications (MB/iter)			Accuracy (%)
		Paillier [3]	Ours	vs. Paillier	Paillier [3]	Ours	vs. Paillier	
HeteroLR [25]	MNIST [46]	275.3	19.9	↑ 14×	234	20.6	↑ 11×	96.1
	Ionosphere [47]	4.6	0.15	↑ 30×	0.76	0.05	↑ 15×	90.0
	Breast Cancer [48]	4.7	0.18	↑ 26×	1.16	0.09	↑ 13×	96.6
VFLwC [34]	MNIST [46]	286.4	20.4	↑ 14×	234	20.6	↑ 11×	96.0
	Ionosphere [47]	4.4	0.16	↑ 28×	0.76	0.05	↑ 15×	89.7
	Breast Cancer [48]	4.1	0.15	↑ 27×	1.16	0.09	↑ 13×	97.1
SecureBoost [35]	MNIST [46]	192.6	47.8	↑ 4×	8.8	0.73	↑ 12×	98.5
	Ionosphere [47]	2.6	1.5	↑ 1.7×	0.26	0.02	↑ 13×	98.2
	Breast Cancer [48]	2.6	1.8	↑ 1.4×	0.41	0.03	↑ 14×	98.4
VF^{2}Boost [7]	MNIST [46]	80.5	22.9	↑ 4×	8.8	0.73	↑ 12×	98.5
	Ionosphere [47]	1.5	0.34	↑ 4×	0.26	0.02	↑ 13×	96.8
	Breast Cancer [48]	1.3	0.45	↑ 4×	0.41	0.03	↑ 14×	98.6

cation scenario for blockchain, and various digital currencies [8], [50]–[52] have also been proposed in recent years.

In order to achieve the confidentiality of the transaction, a typical approach is to commit the balance and the transfer amount with a global homomorphic commitment scheme (e.g., Pedersen commitment [53]) and then derive a secret from hidden coincidences to prove the correctness of the transaction and authorize the transfer.

AHE schemes can be viewed as a computationally hiding and perfectly binding commitment, in which the secret key serves as a natural trapdoor to recovering the message. Meanwhile, EC-based AHE schemes [8], [12] are generally used here to ensure the efficiency of the digital currency system while reducing the communication overhead during the transfer process. In order to ensure efficient decryption, the plaintext length is generally set to about 32 bits in previous works [8], [9], [51]. However, when we deployed a blockchain financial system, we found that 32 bits were not enough because 6 to 8 bits were needed to represent the fractional part. After our actual test, 48 bits is a sufficient plaintext length for a blockchain financial system in our implementation. Unfortunately, existing EC-based AHE schemes are highly time-consuming to decrypt when the plaintext length $\ell = 48$. According to Table II, Exp-ElGamal [12] and Twisted-ElGamal [8] require about 16202 ms and 16195 ms to execute decryption when $\ell = 48$, respectively.

TABLE VII
DECRYPTION EFFICIENCY AND MEMORY ABOUT EXP-ELGAMAL COMBINED WITH FastECDLP WHEN $\ell = 48$ (UNIT: MS, AVERAGE OF 1000 RUNS). T DENOTES THE NUMBER OF THREADS.

ℓ_2	ℓ_1	T = 1	T = 4	T = 16	Memory (GB)
17	31	58.50	23.91	19.05	11.046
18	30	109.22	46.14	36.25	5.526
19	29	207.81	84.88	59.78	2.769
20	28	398.10	162.13	112.84	1.396
21	27	711.63	309.53	222.90	0.721
22	26	1664.74	641.18	452.01	0.407
23	25	3394.97	1295.76	896.575	0.298

We present the experimental results of FastECDLP combined with Exp-ElGamal when $\ell = 48$ in Table VII. We show our experimental results as much as possible and hope readers can better apply FastECDLP in a blockchain system requiring EC-based AHE schemes. According to the hardware configuration, ℓ_1 and ℓ_2 can be configured according to Table VII to balance decryption efficiency and memory space. For example, if the blockchain node is lightweight and can only run a maximum of 2 GB of memory and 4 threads, $\ell_1 = 28$ and $\ell_2 = 20$ are the optimal configurations.

PGC [8] is a decentralized confidential payment system with audibility, and the AHE scheme used is Twisted-ElGamal with $\ell = 32$. We compare the decryption efficiency of Exp-ElGamal combined with FastECDLP and Twisted-ElGamal with a single thread at $\ell = 32$ in Table VIII. It shows that the advantages of FastECDLP become more pronounced as ℓ_2 increases, and the decryption efficiency can be improved by at most 20 times.

TABLE VIII
DECRYPTION EFFICIENCY OF EXP-ELGAMAL COMBINED WITH FastECDLP AND TWISTED-ELGAMAL [8] WITH A SINGLE THREAD AT $\ell = 32$ (UNIT: MS, AVERAGE OF 1000 RUNS).

ℓ_2	ℓ_1	Twisted-ElGamal [8]	Ours	vs. Twisted-ElGamal
7	25	1.878	0.148	↑ 13×
8	24	2.891	0.213	↑ 13×
9	23	4.375	0.330	↑ 13×
10	22	8.147	0.522	↑ 16×
11	21	14.753	0.867	↑ 17×
12	20	30.520	1.665	↑ 18×
13	19	57.488	3.148	↑ 18×
14	18	124.227	6.437	↑ 20×

As for the memory overhead, we also compare the memory of Exp-ElGamal combined with FastECDLP and Twisted-ElGamal at $\ell = 32$ in Table IX. Twisted-ElGamal [8] uses the point compression technique from [24]. Therefore, an element in $\mathbf{T}_1$ of Twisted-ElGamal theoretically occupies only 64 bits of memory, the same as optimized $\mathbf{T}'_1$ at a single element. However, the sizes of $\mathbf{T}_1$ and $\mathbf{T}'_1$ are 2^{ℓ_1} and $2^{\ell_1 - 1}$, respectively. Moreover, our cuckoo hashing can reduce the memory overhead by 5 times due to the redundancy of the hashmap. Therefore, we should reduce the memory overhead by more than 10 times.

According to Table IX, Twisted-ElGamal require 0.953 GB of memory when $\ell_2 = 8$, and we need only 0.086 GB, which reduces memory overhead by about 11 times, which also confirms our above analysis.

TABLE IX
MEMORY OVERHEAD OF EXP-ELGAMAL COMBINED WITH FastECDLP AND TWISTED-ELGAMAL [8] AT $\ell = 32$ (UNIT: GB).

ℓ_2	ℓ_1	Twisted-ElGamal [8]	Ours	vs. Twisted-ElGamal
7	25	1.952	0.172	$\uparrow 11\times$
8	24	0.953	0.086	$\uparrow 11\times$
9	23	0.478	0.043	$\uparrow 11\times$
10	22	0.246	0.021	$\uparrow 11\times$
11	21	0.126	0.010	$\uparrow 11\times$
12	20	0.063	0.006	$\uparrow 12\times$
13	19	0.036	0.003	$\uparrow 12\times$
14	18	0.015	0.001	$\uparrow 12\times$

When the computation or memory overhead is constant, we compare Exp-ElGamal combined with FastECDLP and Twisted-ElGamal in Table X. If the required memory is almost equal (0.015 GB and 0.010 GB), then the decryption of Twisted-ElGamal and Exp-ElGamal combined with FastECDLP cost 124.227 ms and 0.867 ms, respectively. If the computation overhead is almost equal (1.878 ms and 1.665 ms), then Twisted-ElGamal and Exp-ElGamal combined with FastECDLP require 1.952 GB and 0.006 GB of memory, respectively. In other words: If the memory overhead is constant, then we can improve the decryption efficiency by 140 times, and if the computation overhead is constant, then we can reduce the memory overhead by 325 times.

TABLE X
COMPARISON OF EXP-ELGAMAL COMBINED WITH FastECDLP AND TWISTED-ELGAMAL [8] AT $\ell = 32$ WHEN THE COMPUTATION OR MEMORY OVERHEAD IS CONSTANT.

Constant	Schemes	Memory (GB)	Runtime (ms)	vs. [8]
Memory	[8]	0.015	124.227	$\uparrow 140\times$
	Ours	0.010	0.867	
Runtime	[8]	1.952	1.878	$\uparrow 325\times$
	Ours	0.006	1.665	

To sum up, FastECDLP also has solid practical significance in the blockchain payment system.

C. Application to online auctions

Many online auction systems offer a service to customers that one can submit a maximum bid to the system. Obviously, a maximum bid is private information. Moreover, all participants should conduct online auctions fairly. Therefore, a Secure Comparison (SC) protocol is usually used here. SC is an important problem in multi-party computation, and it involves the comparison of two or more secret values in a privacy-preserving manner. There are many SC protocols [10], [54], [55] based on the AHE schemes. Meanwhile, RSA-based AHE schemes cause a lot of communication overhead in frequent bidding comparisons due to the long ciphertext. Therefore, EC-based AHE schemes also has certain advantages in the online auction scenario.

Damgård et al. [10] set $\ell = 16$ in their online auction system, and only conducted experiments at 80-bit security level. That is, the bid should be less than 65536, which can easily cause bid overflow. To ensure the bid cannot overflow,

TABLE XI
DECRYPTION EFFICIENCY AND MEMORY ABOUT EXP-ELGAMAL COMBINED WITH FastECDLP WHEN $\ell = 40$ (UNIT: MS, AVERAGE OF 1000 RUNS). T DENOTES THE NUMBER OF THREADS.

ℓ_2	ℓ_1	T = 1	T = 4	T = 16	Memory (GB)
9	31	0.254	0.142	0.135	11.044
10	30	0.458	0.272	0.242	5.522
11	29	0.833	0.385	0.330	2.761
12	28	1.612	0.702	0.536	1.381
13	27	3.076	1.365	1.018	0.690
14	26	6.294	3.822	2.421	0.345
15	25	13.273	5.610	4.617	0.173
16	24	24.369	11.631	7.977	0.087
17	23	45.701	21.689	15.454	0.045
18	22	93.257	40.344	30.273	0.025
19	21	175.123	177.700	59.834	0.018
20	20	351.186	153.452	111.559	0.021
21	19	721.261	308.065	223.793	0.034

we set $\ell = 40$ bits, i.e., the maximum bid is about one trillion. We apply Exp-ElGamal [12] combined with FastECDLP to the online auction system in [10]. We present the experimental results at 128-bit security level when $\ell = 40$ in Table XI. Similar to the blockchain payment system, ℓ_1 and ℓ_2 can be set according to the needs of the auction system to balance the computation and memory overhead.

VII. CONCLUSION

In this paper, we design an efficient algorithm to solve the small exponential ECDLP for EC-based AHE schemes. We combine the BSGS algorithm, the tree-based Montgomery's trick, and cuckoo hashing to form the basic architecture of FastECDLP and perform deep optimizations from two points: computation and memory overhead. Furthermore, we provide a concrete implementation and fully report the experimental results. We try our best to ensure efficient decryption while maximizing the plaintext length of EC-based AHE schemes. As far as we know, our work is state-of-the-art in solving the small exponential ECDLP for EC-based AHE schemes. On this issue, we welcome further research. We strongly expect that the plaintext length of EC-based AHE schemes can be further extended (e.g., 64 bits) with acceptable decryption efficiency, which is also the direction of our future efforts.

REFERENCES

[1] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "paillier99," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.500.3989&rep=rep1&type=pdf
[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*. ACM, 2009, pp. 169–178.
[3] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1592. Springer, 1999, pp. 223–238.
[4] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 19–38.
[5] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *CCS*. ACM, 2017, pp. 619–631.
[6] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning," in *USENIX Annual Technical Conference*. USENIX Association, 2020, pp. 493–506.

[7] F. Fu, Y. Shao, L. Yu, J. Jiang, H. Xue, Y. Tao, and B. Cui, "Vf^{2}boost: Very fast vertical federated gradient boosting for cross-enterprise learning," in *SIGMOD Conference*. ACM, 2021, pp. 563–576.

[8] Y. Chen, X. Ma, C. Tang, and M. H. Au, "PGC: decentralized confidential payment system with auditability," in *ESORICS (1)*, ser. Lecture Notes in Computer Science, vol. 12308. Springer, 2020, pp. 591–610.

[9] S. Ma, Y. Deng, D. He, J. Zhang, and X. Xie, "An efficient NIZK scheme for privacy-preserving transactions over account-model blockchain," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 2, pp. 641–651, 2021.

[10] I. Damgård, M. Geisler, and M. Krøigaard, "Efficient and secure comparison for on-line auctions," in *ACISP*, ser. Lecture Notes in Computer Science, vol. 4586. Springer, 2007, pp. 416–430.

[11] W. Gao, W. Yu, F. Liang, W. G. Hatcher, and C. Lu, "Privacy-preserving auction for big data trading using homomorphic encryption," *IEEE Trans. Netw. Sci. Eng.*, vol. 7, no. 2, pp. 776–791, 2020.

[12] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1233. Springer, 1997, pp. 103–118.

[13] D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *TCC*, ser. Lecture Notes in Computer Science, vol. 3378. Springer, 2005, pp. 325–341.

[14] F. Tang, G. Ling, and J. Shan, "Additive homomorphic encryption schemes based on sm2 and sm9," *Journal of Cryptologic Research*, vol. 9, no. 3, pp. 535–549, 2022.

[15] T. Okamoto and S. Uchiyama, "A new public-key cryptosystem as secure as factoring," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1403. Springer, 1998, pp. 308–318.

[16] M. Joye and B. Libert, "Efficient cryptosystems from 2 ^{k} -th power residue symbols," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 7881. Springer, 2013, pp. 76–92.

[17] H. Ma, S. Han, and H. Lei, "Optimized paillier's cryptosystem with fast encryption and decryption," in *ACSAC*. ACM, 2021, pp. 106–118.

[18] E. Barker, E. Barker, W. Burr, W. Polk, M. Smid *et al.*, *Recommendation for key management: Part 1: General*. National Institute of Standards and Technology, Technology Administration, 2006.

[19] D. Shanks, "Class number, a theory of factorization, and genera," in *Proc. of Symp. Math. Soc., 1971*, vol. 20, 1971, pp. 41–440.

[20] J. M. Pollard, "Monte carlo methods for index computation (mod) p," *Mathematics of computation*, vol. 32, no. 143, pp. 918–924, 1978.

[21] S. D. Galbraith and R. S. Ruprai, "Using equivalence classes to accelerate solving the discrete logarithm problem in a short interval," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, vol. 6056. Springer, 2010, pp. 368–383.

[22] D. J. Bernstein and T. Lange, "Computing small discrete logarithms faster," in *INDOCRYPT*, ser. Lecture Notes in Computer Science, vol. 7668. Springer, 2012, pp. 317–338.

[23] F. Zhang and S. Liu, "Solving ECDLP via list decoding," in *ProvSec*, ser. Lecture Notes in Computer Science, vol. 11821. Springer, 2019, pp. 222–244.

[24] P. Chatzigiannis, K. Chalkias, and V. Nikolaenko, "Homomorphic decryption in blockchains via compressed discrete-log lookup tables," in *DPM/CBT@ESORICS*, ser. Lecture Notes in Computer Science, vol. 13140. Springer, 2021, pp. 328–339.

[25] S. Hardy, W. Henecka, H. Ivey-Law *et al.*, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017. [Online]. Available: https://arxiv.org/abs/1711.10677

[26] M. Shen, X. Tang, L. Zhu, X. Du, and M. Guizani, "Privacy-preserving support vector machine training over blockchain-based encrypted iot data in smart cities," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 7702–7712, 2019.

[27] Q. Wang, J. Huang, Y. Chen, C. Wang, F. Xiao, and X. Luo, "*PROST*: Privacy-preserving and truthful online double auction for spectrum allocation," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 2, pp. 374–386, 2019.

[28] B. Jia, X. Zhang, J. Liu, Y. Zhang, K. Huang, and Y. Liang, "Blockchain-enabled federated learning data protection aggregation scheme with differential privacy and homomorphic encryption in iiot," *IEEE Trans. Ind. Informatics*, vol. 18, no. 6, pp. 4049–4058, 2022.

[29] C. Chen, J. Zhou, L. Wang, X. Wu, W. Fang, J. Tan, L. Wang, A. X. Liu, H. Wang, and C. Hong, "When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control," in *KDD*. ACM, 2021, pp. 2652–2662.

[30] S. D. Galbraith, P. Wang, and F. Zhang, "Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm," *Adv. Math. Commun.*, vol. 11, no. 3, pp. 453–469, 2017.

[31] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy, "Secure sharing of partially homomorphic encrypted iot data," in *SenSys*. ACM, 2017, pp. 29:1–29:14.

[32] P. K. Mishra, "Efficient simultaneous inversion in parallel and application to point multiplication in ecc," in *International Conference on Information Security and Cryptology*. Springer, 2005, pp. 324–335.

[33] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[34] S. Yang, B. Ren, X. Zhou *et al.*, "Parallel distributed logistic regression for vertical federated learning without third-party coordinator," in *Proceedings of the IJCAI'19 Workshop*, 2019.

[35] K. Cheng, T. Fan, Y. Jin *et al.*, "Secureboost: A lossless federated learning framework," *IEEE Intelligent Systems*, vol. 36, no. 6, pp. 87–98, 2021.

[36] K. Matsuo, J. Chao, and S. Tsujii, "An improved baby step giant step algorithm for point counting of hyperelliptic curves over finite fields," in *ANTS*, ser. Lecture Notes in Computer Science, vol. 2369. Springer, 2002, pp. 461–474.

[37] P. Gaudry and É. Schost, "A low-memory parallel version of matsuo, chao, and tsujiis algorithm," in *ANTS*, ser. Lecture Notes in Computer Science, vol. 3076. Springer, 2004, pp. 208–222.

[38] Z. Gao, L. Xu, and W. Shi, "Mapreduce for elliptic curve discrete logarithm problem," in *SERVICES*. IEEE Computer Society, 2016, pp. 39–46.

[39] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of Annual International Cryptology Conference (CRYPTO)*. Springer, 1984, pp. 10–18.

[40] *IT Security techniques-Encryption algorithms-Part 6: Homomorphic encryption*. ISO/IEC 18033-6:2019.

[41] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai, "Efficient circuit-based PSI with linear communication," in *EUROCRYPT (3)*, ser. Lecture Notes in Computer Science, vol. 11478. Springer, 2019, pp. 122–153.

[42] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[43] [Online]. Available: https://github.com/btcsuite/btcd/tree/master/btcec

[44] D. R. Brown, "Sec 2: Recommended elliptic curve domain parameters," *Standars for Efficient Cryptography*, 2010.

[45] J. Konečnỳ, McMahan *et al.*, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016. [Online]. Available: https://arxiv.org/abs/1610.05492

[46] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[47] V. G. Sigillito, S. P. Wing, L. V. Hutton, and K. B. Baker, "Classification of radar returns from the ionosphere using neural networks," *Johns Hopkins APL Technical Digest*, vol. 10, no. 3, pp. 262–266, 1989.

[48] A. Asuncion and D. Newman, "Uci machine learning repository," 2007.

[49] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[50] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[51] P. Chatzigiannis and F. Baldimtsi, "Miniledger: Compact-sized anonymous and auditable distributed payments," in *ESORICS (1)*, ser. Lecture Notes in Computer Science, vol. 12972. Springer, 2021, pp. 407–429.

[52] A. Tomescu, A. Bhat, B. Applebaum, I. Abraham, G. Gueta, B. Pinkas, and A. Yanai, "UTT: decentralized ecash with accountable privacy," *IACR Cryptol. ePrint Arch.*, p. 452, 2022.

[53] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 576. Springer, 1991, pp. 129–140.

[54] I. F. Blake and V. Kolesnikov, "Strong conditional oblivious transfer and computing on intervals," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 3329. Springer, 2004, pp. 515–529.

[55] T. Veugen, "Encrypted integer division and secure comparison," *Int. J. Appl. Cryptogr.*, vol. 3, no. 2, pp. 166–180, 2014.