mathematics

MDPI

Article

Efficient Implementations of Sieving and Enumeration Algorithms for Lattice-Based Cryptography

Hami Satılmış [1], Sedat Akleylek [1,*] and Cheng-Chi Lee [2,3,*]

[1] Department of Computer Engineering, Ondokuz Mayıs University, Samsun 55200, Turkey; hami.satilmis@bil.omu.edu.tr
[2] Department of Library and Information Science, Research and Development Center for Physical Education, Health, and Information Technology, Fu Jen Catholic University, New Taipei City 242, Taiwan
[3] Department of Photonics and Communication Engineering and Department of Computer Science and Information Engineering, Asia University, Taichung 413, Taiwan
* Correspondence: sedat.akleylek@bil.omu.edu.tr (S.A.); cclee@mail.fju.edu.tw (C.-C.L.)

Abstract: The security of lattice-based cryptosystems is based on solving hard lattice problems such as the shortest vector problem (SVP) and the closest vector problem (CVP). Various cryptanalysis algorithms such as (Pro)GaussSieve, HashSieve, ENUM, and BKZ have been proposed to solve these hard problems. Several implementations of these algorithms have been developed. On the other hand, the implementations of these algorithms are expected to be efficient in terms of run time and memory space. In this paper, a modular software package/library containing efficient implementations of GaussSieve, ProGaussSieve, HashSieve, and BKZ algorithms is developed. These implementations are considered efficient in terms of run time. While constructing this software library, some modifications to the algorithms are made to increase the performance. Then, the run times of these implementations are compared with the others. According to the experimental results, the proposed GaussSieve, ProGaussSieve, and HashSieve implementations are at least 70%, 75%, and 49% more efficient than previous ones, respectively.

Keywords: lattice-based cryptography; sieving algorithms; efficient software implementations; SVP

Citation: Satılmış, H.; Akleylek, S.; Lee, C.-C. Efficient Implementations of Sieving and Enumeration Algorithms for Lattice-Based Cryptography. *Mathematics* **2021**, *9*, 1618. https://doi.org/10.3390/math9141618

Academic Editor: Luis Hernández Encinas

Received: 2 June 2021
Accepted: 5 July 2021
Published: 8 July 2021

1. Introduction

Traditional public key cryptosystems such as RSA and (EC)DSA are based on the hardness of the integer factorization and the discrete logarithm problem [1]. However, due to the Shor algorithm in [2], they are insecure in the quantum era. For this reason, new cryptosystems are needed to avoid vulnerability in communication networks after the widespread use of quantum computers. The family of lattice-based cryptosystems is one of the candidates in the quantum era due to the efficiency and security reasons [1]. The lattice-based cryptography can be used in different security areas, such as identification and authentication [3,4]. The basis of the difficulty of lattice-based cryptography consists of lattice problems such as SVP and CVP, for which the solution is unknown in polynomial-time, even in the quantum computer era. To solve hard problems such as SVP and CVP, i.e., to break the lattice-based cryptography, many sieving-based and enumeration-based algorithms and their implementations of these algorithms are proposed in [5–10].

1.1. Previous Works

The main idea in the sieving algorithms such as the GaussSieve and the ProGaussSieve is to store the list data structure in memory, where vectors with larger sizes are held, and to continue processing by finding vectors close to the size of the intended shortest vector [9]. The efficiency of sieving algorithms is evaluated according to memory usage and the run times [5,6]. Therefore, sieving algorithms such as the GaussSieve and the ProGaussSieve, working with the same main idea of having only a few structural or technical modifica-

tions, have been proposed in [5,6,10]. In [11], the AKS, the first exponential-time sieving algorithm, has the asymptotic run time complexity $2^{2.465n+o(n)}$ and the space complexity $2^{1.233n+o(n)}$, where n is the lattice size. With the sieving algorithm of Nguyen–Vidick, which is the heuristic version of the AKS algorithm, the implementation of the AKS algorithm was developed in [12]. It was noted that memory problems in the AKS might occur as the size grows in [12]. The ListSieve algorithm in [5], in which the main operation is to start processing with an empty list data structure and to add a shorter vector to the list each time until it contains the shortest vector, operates with the asymptotic run time complexity $2^{3.199n+o(n)}$ and the space complexity $2^{1.325n+o(n)}$. Micciancio and Voulgaris suggested the ListSieve algorithm and proposed the GaussSieve algorithm in [5] run with the ListSieve in the same idea except with a few changes. The GaussSieve algorithm in [5] works asymptotically with the run time $2^{0.48n+o(n)}$ complexity and the space complexity $2^{0.18n+o(n)}$. Applying a progressive approach to the GaussSieve algorithm, Laarhoven and Mariano introduced the ProGaussSieve sieving algorithm in [6], which has the asymptotic run time complexity $2^{0.42n+o(n)}$ and the space complexity $2^{0.21n+o(n)}$. The HashSieve sieving algorithm in [10] was revealed by carrying out Charikar's angular locality-sensitive hash (angular-LSH) family [13] to the GaussSieve algorithm, was proven to be faster than the GaussSieve algorithm, and found the shortest vector in the asymptotic run time complexity $2^{0.3366n+o(n)}$ and the space complexity $2^{0.3366n+o(n)}$. In [14], a new algorithm was proposed by combining the k-means LSH function with the HashSieve.

The standard implementation of the GaussSieve algorithm was first developed by Micciancio and Voulgaris in [5] by using the NTL library [15]. In 2014, the first known parallel implementation of the GaussSieve algorithm was developed in [16]. The parallel version of the GaussSieve algorithm in [17] was implemented using the distributed-memory method on a CPU. Later, another parallel GaussSieve implementation using enhanced lock-free list data structure was introduced in [18]. Using the parallel GaussSieve algorithm method of Ishugura et al., the parallel implementation of the GaussSieve algorithm was developed on a GPU in [19]. The implementation of the ProGaussSieve algorithm, the progressive version of the GaussSieve algorithm, was developed by Laarhoven and Mariano in [6]. In 2015, the first known standard version of the HashSieve algorithm was implemented by Laarhoven in [10]. In [20], an SVP solver by using the Voronoi cell of a lattice was implemented. They gave a real performance of this method, although there are some limitations such as increasing memory requirements with the number of dimensions.

The enumeration algorithms are designed to enumerate all lattice points (vectors) in a bounded area to have a solution (finding the shortest vector) with better memory requirements. The first examples of enumeration algorithms are Kannan [21], Fincke and Pohst [22], and Schnorr and Euchner [7]. In 2010, the enumeration algorithms were made more efficient by using the "extreme pruning" technique in [23]. Dağdelen and Schneider introduced the parallel implementation of the enumeration algorithm in [24]. In 2016, the parallel implementation of Schnorr and Euchner's enumeration algorithm SE++ was revealed in [25].

The sieving and the enumeration algorithms need to perform reduction algorithms such as the Gram–Schmidt [26], the LLL [26], and the BKZ [8,27] to reduce the size of the lattices before starting their basic operations. The LLL algorithm has several usages in cryptography. For instance, the LLL has an important role in solving the knapsack problem efficiently, the integer factorization [28]. The LLL algorithm and the enumeration algorithms are used as subprocesses in the BKZ algorithm, which is the block width version of the LLL reduction algorithm [29]. The BKZ algorithm provides the most efficient results among the reduction algorithms, which was first introduced by Schnorr in [30], and its implementation was developed in [8]. The BKZ 2.0 version of the BKZ algorithm, which uses methods such as pruning enumeration [23], early termination, and progressive reduction, was introduced and implemented in [31]. In 2014, the parallel implementation of the BKZ algorithm was revealed in [32]. Later, the ACBKZ algorithm and its implementation were

introduced in [33], which is the version of the BKZ algorithm that operates in different blocks in parallel.

1.2. Motivation and Contribution

Sieving, enumeration, and reduction algorithms consist of many common components. Therefore, these components are needed for the implementations of sieving, enumeration, and reduction algorithms. However, the implementations of sieving, enumeration, and reduction algorithms can be efficiently developed by using a software library that includes common components. Note that there is a lack of a modular software library that is used as the infrastructure in developing efficient implementations of these algorithms [34]. In this paper, a modular structured software library is designed to be used as the infrastructure while developing implementations of these algorithms to fill this gap. The developed implementation is designed to be efficient in terms of the run time and the space complexity. In addition to using a modular software infrastructure library, the performance of implementations can be increased through implementation-based improvements that can be made on the algorithms [6]. Furthermore, the proposed modular software library and the efficient implementations can also be used for efficient implementations of other lattice-based schemes. The contributions of this paper are as follows:

- The modular software infrastructure library is developed to be used as an infrastructure to have the efficient implementations of the algorithms.
- With the modular software library containing the commonly used components in the algorithms, the efficient implementations of GaussSieve and ProGaussSieve are provided.
- In order to achieve performance improvements in the implementations of the GaussSieve and the ProGaussSieve algorithms, it is proposed to make changes to the termination criterion of these algorithms.
- By making novel modifications to the HashSieve implementation developed by Laarhoven [10] and by using the modular software infrastructure library, a faster implementation of HashSieve is achieved.
- Efficient implementations of the ENUM and the BKZ algorithms are developed using the modular software infrastructure library.
- The proposed solution in [8] for the zero vector problem encountered in the LLLFP algorithm (a subprocess in the BKZ) is implemented in the LLLFP module.

The efficient implementations of GaussSieve and ProGaussSieve are compared with the GaussSieve and the ProGaussSieve implementations in [6] with regard to the asymptotic run time complexities. For the same lattice samples, HashSieve implementations are compared with that in [10]. In addition, the accuracy of the outputs of the efficient implementations of ENUM and BKZ are checked by comparing them with the outputs of the SageMath [35].

1.3. Organization

The rest of this paper is organized as follows. In Section 2, the mathematical background of the lattice-based cryptosystems; the lattice algorithms; and the GaussSieve, the ProGaussSieve, the HashSieve, the ENUM, and the BKZ algorithms are recalled with possible improvements. The software features of the modular software infrastructure library used to develop the efficient implementations of these algorithms are mentioned in Section 3. In addition to this section, the efficient implementations of these algorithms are detailed with novel modifications. In Section 4, the run times of efficient implementations are compared with those of the implementations in the literature. Finally, the results obtained in this paper and the future studies are given in Section 5.

2. Preliminaries

In this section, the main computationally hard problems in the lattice-based cryptography are recalled. Then, the common subcomponents and the algorithms developed to solve these hard problems are mentioned. Finally, the basic working order of the algorithms

developed for solving hard problems is explained through the pseudo-codes. Table 1 shows some special notations that are often used in the mathematical descriptions and the algorithms.

Table 1. Notations.

Notations	Definitions
$\|\|v\|\|$	Euclidean norm of the vector v
$\|x\|$	Absolute value of the number x
$\lceil x \rfloor$	Rounding the number x to the nearest integer
$S \cup \{v\}$	Adding vector v to set S
$S/\{v\}$	Removing the vector v from the set S
v^*	Vector v reduced by Gram-Schmidt
v'	Converting vector $v \in \mathbb{Z}$ to $v \in \mathbb{R}$
cl	Number of collisions in sieving algorithms
T	Hash tables in the HashSieve algorithm
β	Local lattices size in the BKZ algorithm
δ	Floating-point error value for the LLLFP algorithm

2.1. Mathematical Background

Mathematical definitions of the lattice structure and the hard problems in these cryptosystems are given below.

Definition 1 (Lattice). *In $\mathbb{R}^n$, the set of points consisting of linear independent integer vectors $\{v_1, \ldots, v_n\}$ of the basis B is called the lattice.*

$$L = L(B) = \left\{ \sum_{i=1}^{n} c_i v_i : c_i \in \mathbb{Z} \right\} \tag{1}$$

Definition 2 (SVP). *Let the shortest lattice vector be $\lambda_1(L) = min||v||$, provided that it is $v \in L$ and $v \neq 0$. SVP is the problem of finding the lattice vector $v \in L$, in $||v|| = \lambda_1(L)$ equality. In other words, SVP is the problem of finding the vector with the shortest Euclidean norm from the lattice basis vectors [36].*

Definition 3 (CVP). *Let $t \in \mathbb{R}^n$ be a target point, and the smallest distance between the lattice vector v and t is defined as $d(t, L) := min||v - t||$. CVP is the problem of finding the lattice vector $v \in L$, in $||v|| = d(t, L)$ equality. In other words, CVP is the problem of finding the lattice vector closest to the chosen target point [37].*

2.2. Common Submodules in Algorithms

Many mathematical operation structures and algorithms are common in the sieving, enumeration, the BKZ reduction algorithms. These common mathematical operation structures and algorithms, called common submodules, are carried out as subprocesses in the sieving, enumeration, and BKZ reduction algorithms. Among these common algorithms, the GaussReduce reduction [5], the Gram–Schmidt reduction [26], the LLL reduction [26], the LLLFP reduction [8], and the Klein's Nearest Neighbor [38] are used for the following purposes and operations. In addition, the mathematical operation structures that are commonly used with algorithms are given below.

- **The GaussReduce Algorithm:** This reduction algorithm is used in the Gauss-based sieving algorithms to reduce the size of a lattice vector by other lattice vectors.

- **The Gram–Schmidt Algorithm:** This reduction algorithm obtains the Gram–Schmidt constants and the reduced lattice consisting of vectors perpendicular to each other as much as possible. The resulting reduced lattices and constants are given as input parameters to the sieving, enumeration, and reduction algorithms such as the BKZ and the LLL.
- **The LLL and The LLLFP Algorithms:** These algorithms produce lattices consisting of vectors orthogonal to each other and that are reduced. The reduced lattice is given as an input parameter to the sieving algorithms and is used as a subprocess in the BKZ. The only difference between the LLLFP algorithm and the LLL algorithm is that the LLLFP algorithm minimizes the floating-point errors in the LLL algorithm.
- **The Klein's Nearest Neighbor Algorithm:** It produces new sample vectors to be used in the sieving algorithms. This new sample vector is used in the reduction operations in the sieving algorithms. The Klein's algorithm is composed of the nearA algorithm [38] as the main algorithm and the Randomized Rounding algorithm [39] as a submodule.
- **The Mathematical Operations:** Vector arithmetic is needed in the sieving, the enumeration, the BKZ, and the subcomponents of these algorithms such as the GaussReduce and the Gram–Schmidt submodules. These operations are the Euclidean norm of a vector, the vector addition/subtraction, and the inner product.

2.3. The GaussSieve and the ProGaussSieve Algorithms

The main idea in the GaussSieve algorithm, solving the shortest vector problem, is to add shorter new lattice vectors for each iteration to the list data structure. In addition, the GaussSieve algorithm is to reduce the new lattice vector with the list vectors while at the same time reducing all list vectors with the new lattice vector.

The reduced lattice basis B and the termination criterion c (the total number of collisions) are sent to the GaussSieve algorithm as the inputs. The GaussSieve algorithm first takes a vector v from the stack data structure S where the sample vectors are stored or generates a new sample vector v in the Klein's Nearest Neighbor algorithm (line 5 in Algorithm 1). The new sample vector v is given as input to the GaussReduce reduction algorithm. In this reduction algorithm, the vector v is reduced by using all list vectors w (line 6 in Algorithm 1). The reduced new sample vector v reduces all list vectors w if the vector v satisfies the conditions (line 7 in Algorithm 1). Later, the GaussSieve algorithm compares the Euclidean norm of the reduced new sample vector v with the Euclidean norm of its previous state (line 9 in Algorithm 1). As a result of the comparison, if there is no change in the length of the new sample vector v, the GaussSieve algorithm adds the vector v to the list data structure L (line 10 in Algorithm 1). If the reduced new sample vector v has a change in length and its length is different from zero, the GaussSieve algorithm adds the reduced vector v to the stack data structure S (line 13 in Algorithm 1). If the reduced new sample vector v has a change in length (the length is zero), the GaussSieve algorithm increases the number of collisions cl by one (line 15 in Algorithm 1). The GaussSieve algorithm, which consists of these process steps in general, iterates all of these steps and terminates operations when it reaches (the termination criterion c) a certain number of collisions cl (zero vector state). The GaussSieve algorithm, which terminates the process, outputs the shortest lattice vector in the list L. In Algorithm 1, which shows the pseudo-code of the ProGaussSieve algorithm, if the lines, written in red, are removed, the pseudo-code of the GaussSieve algorithm is given with slight modifications.

The ProGaussSieve algorithm, a different version of the Gauss-based sieving algorithm, operates in the same manner and procedure as in the GaussSieve algorithm. The main difference between the ProGaussSieve algorithm and the GaussSieve algorithm is that the ProGaussSieve algorithm divides the lattice into smaller subparts and starts operating with the smallest one, although the GaussSieve algorithm starts operating on the whole lattice.

Algorithm 1 ProGaussSieve Algorithm [6].

Input: Reduced lattice basis B, termination criterion c and $progressive = min\{10, n\}$ constant
Output: The shortest vector on the basis of lattice B

$L = \emptyset$
$S = \emptyset$
$cl = 0$
while true **do**
  Taking a new vector v sampled from Klein's algorithm or the stack S
  $GaussReduce(v, w \in L)$
  $GaussReduce(w \in L, v)$
  Move the reduced vectors $w \in L$ from list L to the stack S
  if v has not changed **then**
    $L = L \cup \{v\}$
  else
    if $v \neq 0$ **then**
      $S = S \cup \{v\}$
    else
      $cl = cl + 1$
      if $cl = c$ **then**
        if $progressive = n$ **then**
          return $argmin_{v \in L} ||v||$
        else
          $progressive{+}{+}$
          $cl = 0$

The ProGaussSieve algorithm in Algorithm 1 takes the constant *progressive* as an input parameter, which determines the size of the lattice subparts, as well as the input parameters of the GaussSieve algorithm. The ProGaussSieve algorithm starts the operation by dividing the lattice into subparts according to the constant *progressive* and does the same operations as the GaussSieve algorithm on the smallest lattice subpart. When the ProGaussSieve algorithm reaches a certain number of collisions cl (zero vector state), it increases the constant *progressive*. Then, the ProGaussSieve algorithm continues to operate on the following lattice subpart. After the ProGaussSieve algorithm operating over the whole lattice and the number of collisions cl reaches the value of the termination criterion c, it halts the operation and outputs the shortest vector of the lattice.

2.4. The HashSieve Algorithm

The main idea and the operation procedure in the HashSieve algorithm, which uses Charikar's angular-LSH (locality sensitive hashing) family, is almost the same as the GaussSieve algorithm. The GaussSieve algorithm stores the vectors and uses them to reduce in a list data structure. In contrast, the HashSieve algorithm stores the vectors and uses them to reduce the hash tables and the list data structure. Note that LSH is not a cryptographic hash function approach and is helpful to see whether the distance between vectors are small enough.

In the angular-LSH family, given a target vector v and a hash vector a, the hash value consists of a single bit $h_a(v) \in \{0, 1\}$ and is calculated as

$$h_a(v) = \left\{ \begin{array}{lll} 1 & \text{if} & \langle a, v \rangle \geq 0; \\ 0 & \text{if} & \langle a, v \rangle < 0. \end{array} \right\} \tag{2}$$

The angular-LSH family consists of functions $\mathcal{H} = \{h_a\}$ in the randomly drawn $a \in \mathbb{R}^n$ from an n-dimensional Gaussian distribution. These hash functions have the property that vectors mapped to the same bucket have a higher probability of being closer than the "average" list vectors [40].

The pseudo-code of the HashSieve algorithm, which receives the reduced lattice B and the termination criterion c (the total number of collisions) as input parameters, is given in Algorithm 2. The HashSieve algorithm starts to operate with the empty hash tables T and the empty stack data structure S containing the sample vectors. In each iteration, the HashSieve algorithm first takes either the sample vector v from stack S or the new sample vector v generated by the Klein's Nearest Neighbor algorithm (line 5 in Algorithm 2). The HashSieve algorithm reduces the vector v to the closest candidate vectors in hash tables T. Then, the HashSieve algorithm reduces the vector w by using the reduced vector v (line 10 in Algorithm 2). After moving the nonzero reduced vectors w to stack S, the HashSieve algorithm inserts the nonzero reduced vector v into the hash tables T at the end of the iteration (line 16 and 20 in Algorithm 2). If the reduced vector v is a zero vector, the HashSieve algorithm increases the collision number cl (line 18 in Algorithm 2). The HashSieve algorithm, which iteratively performs these operations until the collision number cl reaches the termination criterion c, gives the shortest vector of the lattice as an output.

Algorithm 2 HashSieve Algorithm [10].

Input: Reduced lattice basis B, termination criterion c
Output: The shortest vector on the basis of lattice B

1: $S = \emptyset$
2: $cl = 0$
3: T empty hash tables $H_1, \ldots, H_T$
4: **while** $cl < c$ **do**
5: Taking a new vector v sampled from Klein's algorithm or the stack S
6: **while** $\exists w \in H_1[h_1(v)], \ldots, H_T[h_T(v)] : ||v \pm w|| < ||v||$ **do**
7: **for each** *Hash table* $H_i, \ldots, H_T$ **do**
8: Find the closest candidate vectors $C = H_i[h_i(v)]$
9: **for each** $w \in C$ **do**
10: Reduce vector v with vector w and reduce vector w with vector v
11: **if** w has changed **then**
12: $H_i = H_i / \{w\}$
13: **if** $w = 0$ **then**
14: $cl = cl + 1$
15: **else**
16: $S = S \cup \{w\}$
17: **if** $v = 0$ **then**
18: $cl = cl + 1$
19: **else**
20: $H_i = H_i \cup \{v\}$

2.5. The ENUM Algorithm

Although the primary purpose of the ENUM algorithm working with the enumeration idea is to solve the SVP problem, it is intended to be used as a subprocess in Schnorr and Euchner's BKZ reduction algorithm. The ENUM enumeration algorithm is used to calculate the smallest area of the local lattice blocks found in the BKZ algorithm.

The ENUM algorithm, the pseudo-code of which is given in Algorithm 3 and developed specifically for Schnorr and Euchner's BKZ algorithm, takes the indices j and k as input parameters. For the smallest $\tilde{c}_j$ in the function c_j, the ENUM algorithm performs an in-depth search on all integer vectors $\{\tilde{u}_t, \ldots, \tilde{u}_k\}$, providing the condition $\tilde{c}_j > c_t(\tilde{u}_t, \ldots, \tilde{u}_k)$ (between 7 and 17 lines in Algorithm 3). The ENUM algorithm, which calculates the operation $\tilde{c}_t = c_t(\tilde{u}_t, \ldots, \tilde{u}_k)$, assigns the value 1 to the Δ_t and variables $\tilde{u}_t$ at the level t (lines 23 and 24 in Algorithm 3). The algorithm, which always assigns the maximum value of t to s, assigns one of the sequential values $1, -1, 2, -2, 3, -3, \ldots$ to the variable Δ_t when the condition $\tilde{c}_t \geq \bar{c}_j$ is satisfied (line 23 in Algorithm 3). The ENUM algorithm, running iteratively, increases the variables s and t by 1 in each iteration (line 19 in Algorithm 3). The

algorithm, which continues to operate at the level $t-1$, assigns the result of the operation $-y_t + \lceil -y_t \rfloor$ to the variable δ_t and the value 0 to the variable Δ_t (line 10 in Algorithm 3). The algorithm, which begins to operate at level t again, sets one of the sequential values $1, -1, 2, -2, 3, -3, \ldots$ or $-1, 1, -2, 2, -3, 3, \ldots$ to the value Δ_t (line 23 in Algorithm 3). The ENUM algorithm, performing iteratively, returns the smallest area $\{u_j, \ldots, u_k\} \in \mathbb{Z}^{k-j+1}$ as an output when it reaches the termination criterion.

Algorithm 3 ENUM Algorithm [7].

Input: j and k for $1 \leq j < k \leq m$
Output: The smallest field $\{u_j, \ldots, u_k\} \in \mathbb{Z}^{k-j+1}$

1: Values $c_i = \|v_i^{*'}\|^2$ and local lattice basis vectors $\{v_j, \ldots, v_k\}$ for $j \leq t < i \leq k$ in the BKZ algorithm
2: $\bar{c}_j = c_j$, $\tilde{u}_j = u_j = 1$, $y_j = \Delta_j = 0$, $s = t = j$, $\delta_j = 1$
3: **for** $i = j+1$ **to** $k+1$ **do**
4: $\quad \tilde{c}_i = u_i = \tilde{u}_i = y_i = \Delta_i = 0$, $\delta_i = 1$
5: **while** $t \leq k$ **do**
6: $\quad \tilde{c}_t = \tilde{c}_{t+1} + (y_t + \tilde{u}_t)^2 c_t$
7: $\quad$ **if** $\tilde{c}_t < \bar{c}_j$ **then**
8: $\quad\quad$ **if** $t > j$ **then**
9: $\quad\quad\quad t = t-1$, $y_t = \sum_{i=t+1}^{s} \tilde{u}_i \mu_{i,t}$
10: $\quad\quad\quad \tilde{u}_t = w_t = \lceil -y_t \rfloor$, $\Delta_t = 0$
11: $\quad\quad\quad$ **if** $\tilde{u}_j > -y_t$ **then**
12: $\quad\quad\quad\quad \delta_t = -1$
13: $\quad\quad\quad$ **else**
14: $\quad\quad\quad\quad \delta_t = 1$
15: $\quad\quad$ **else**
16: $\quad\quad\quad \bar{c}_j = \tilde{c}_j$
17: $\quad\quad\quad u_i = \tilde{u}_i$ for $i = j, \ldots, k$
18: $\quad$ **else**
19: $\quad\quad t = t+1$, $s = max(s, t)$
20: $\quad\quad$ **if** $t < s$ **then**
21: $\quad\quad\quad \Delta_t = -\Delta_t$
22: $\quad\quad$ **if** $\Delta_t \delta_t \geq 0$ **then**
23: $\quad\quad\quad \Delta_t = \Delta_t + \delta_t$
24: $\quad\quad\quad \tilde{u}_t = w_t + \Delta_t$
25: **return** The smallest field $\{u_j, \ldots, u_k\} \in \mathbb{Z}^{k-j+1}$

2.6. The BKZ Algorithm

Schnorr and Euchner's BKZ algorithm is used for lattice reduction. The operation is to perform on local sublattices in which the sizes are determined by the parameter β. In Schnorr and Euchner's BKZ algorithm, in which the reduction quality varies according to parameter β, the LLLFP reduction and the ENUM enumeration algorithms developed by Schnorr and Euchner are used as a subprocess.

In Algorithm 4, the pseudo-code of Schnorr and Euchner's BKZ algorithm is detailed. It takes the basis $B = \{v_1, \ldots, v_n\}$ of the n-sized lattice L, the local lattice size value β, the floating-point error value δ (for the LLLFP reduction algorithm), the Gram–Schmidt constants μ, and the Euclidean norms of the reduced Gram–Schmidt lattice vectors $\|v_1^*\|^2, \ldots, \|v_n^*\|^2$ as input parameters. The BKZ algorithm, as the first operation, uses the LLLFP algorithm (to be $F_c = false$), reduces the basis B, and updates the Gram–Schmidt constants μ (line 4 in Algorithm 4). The algorithm, which reduces the local lattice blocks $B_{[j,min(j+\beta-1,n)]}$ for $j = 1, \ldots, n$ by iteratively them, assigns the index j a value of 1 as the initial value (line 6 in Algorithm 4). The BKZ algorithm, which needs the vector $u = (u_1, \ldots, u_n)$ (the smallest field), operates the lattice $L_{[j,k]}$ in the index $k = min(j+\beta-1, n)$ in the ENUM enumeration algorithm in each iteration so that it finds

the vector (line 9 in Algorithm 4). The algorithm, which sets the latest index of the new local lattice block to $h = min(k+1, n)$, produces the new lattice vector $v^{new} = \sum_{i=j}^{k} u_i v_i$ (line 10 in Algorithm 4) if the condition $\|v_j^*\| > \lambda_1(L_{[j,k]})$ is satisfied and includes this vector between the lattice vectors v_{j-1} and v_j. By including the new lattice vector, the BKZ algorithm forms the set of vectors $(v_1, \ldots, v_{j-1}, v^{new}, v_j, \ldots, v_h)$, sends this set of vectors to the LLLFP algorithm (to be $F_c = true$), and then produces the new reduced local lattice $\{v_1, \ldots, v_h\}$. Finally, it updates the constants μ (line 14 in Algorithm 4). If the condition $\|v_j^*\| > \lambda_1(L_{[j,k]})$ is not satisfied, the BKZ algorithm sends the local lattice block $v_1, \ldots, v_h$ to the LLLFP algorithm and updates the constants μ (line 17 in Algorithm 4). Thus, the algorithm, which produces the LLLFP reduced lattice basis $\{v_1, \ldots, v_h\}$, assigns the value 1 to the index j if none of the enumeration operations are successful when the value of the index j reaches the number n. The BKZ algorithm, which is iteratively performing all of these operations, uses the failed enumeration process counter z as a termination criterion. After reducing the whole lattice and when the termination counter z is also $n-1$, the BKZ algorithm returns the BKZ-reduced lattice $\{v_1, \ldots, v_n\}$ as an output.

Algorithm 4 BKZ Algorithm [8].

Input: Lattice basis $B = \{v_1, \ldots, v_n\} \in \mathbb{Z}^n$, block size $2 < \beta < n$, $\frac{1}{2} < \delta < 1$ with the condition δ, constants μ ve $\|v_1^*\|^2, \ldots, \|v_n^*\|^2$

Output: The BKZ-reduced basis $\{v_1, \ldots, v_n\}$

1: $z = 0$
2: $j = 0$
3: $F_c = false$
4: $LLLFP(v_1, \ldots, v_n, \delta, F_c)$
5: **while** $z < n-1$ **do**
6: $\quad j = (j \, mod(n-1)) + 1$
7: $\quad k = min(j + \beta - 1, n)$
8: $\quad h = min(k+1, n)$
9: $\quad u = ENUM(j, k)$
10: $\quad v^{new} = \sum_{i=j}^{k} u_i v_i \quad \triangleright \|\pi_j(\sum_{i=j}^{k} u_i v_i)\| = \lambda_1(L_{[j,k]})$
11: $\quad$ **if** $\|v_j^*\| > \lambda_1(L_{[j,k]})$ **then**
12: $\quad\quad z = 0$
13: $\quad\quad F_c = true$
14: $\quad\quad LLLFP((v_1, \ldots, v_{j-1}, v^{new}, v_j, \ldots, v_h), \delta, F_c)$
15: $\quad$ **else**
16: $\quad\quad z = z + 1$
17: $\quad\quad LLLFP((v_1, \ldots, v_h), \delta, F_c)$
18: **return** The BKZ-reduced basis $\{v_1, \ldots, v_n\}$

3. Modular Software Library

A modular software library to solve SVP in lattice-based cryptography is developed. This library is structured with a divide-and-conquer approach, i.e., submodules/subcomponents commonly used in the sieving, enumeration, and reduction cryptanalysis algorithms are determined and firstly implemented. Then, the connection/relation of these submodules is defined. Adhering to the algorithmic framework emerging from the analysis, common subcomponents required by the algorithms are added to the software library as modules. Then, these modules are the core parts of the efficient implementations of algorithms. The algorithmic framework in Figure 1 shows the dependency relationship between all algorithms and the submodules needed by the algorithms such as the sieving, the ENUM, and the BKZ. The direction of the arrows in Figure 1 means that the algorithm is a subprocess in the target algorithm. For example, the LLLFP algorithm is used as a subprocess in the BKZ algorithm.

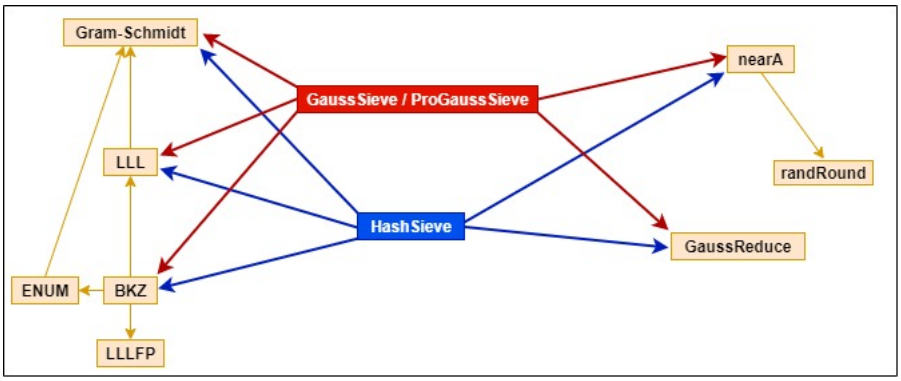

Figure 1. The algorithmic framework of the sieving and enumeration methods.

The software library, developed using the C programming language, includes the Gram–Schmidt, the LLL, the LLLFP, the ENUM enumeration, the Klein's Nearest Neighbor (nearA and Randomized Rounding algorithms), and the GaussReduce algorithms in modules. The software library also contains the modular forms of the vector Euclidean norm calculation, the vector addition/subtraction, and the inner product mathematical operations structures, which are commonly used by all algorithms given in Section 2. These modules, which are used as subcomponents by the efficient implementations of sieving, enumeration, and reduction algorithms, are available in different data types (such as *long long int*, and *double*) in the software library. Since these modules are used as subcomponents in the efficient implementations, they directly affect the run time of the implementations. For this reason, the variables or the parameters in the modules are defined with a structure *pointer* or *double* − *pointer*. Due to the structure *pointer*, the speed of accessing data in the memory during the processing of subcomponents increases. Thus, the run time of efficient implementations is reduced.

Codes 1 and 2 are examples for the modules. The Gram–Schmidt modules that return the reduced lattice output in the data type *double* is given as an example in Code 1. Code 2 shows an example of an inner product module that produces an output of the data type *long long int*.

Code 1. The Gram–Schmidt module.

```
double** GramSchmidt(long long int** B,int N)
{
 double** mu=(double**)malloc(sizeof(double*)*N);
 for(int l=0;l<N;l++){
  mu[l]=(double*)malloc(sizeof(double)*N);
 }
 double** Bs=(double**)malloc(sizeof(double*)*N);
 for(int l=0;l<N;l++){
  Bs[l]=(double*)malloc(sizeof(double)*N);
 }
 int i,j,k;
 for(i=0;i<N;i++){
  for(j=0;j<N;j++){
   mu[i][j]=0;
   Bs[i][j]=(double)B[i][j];
  }
  for(k=0;k<i;k++){
   mu[i][k]=inner_product(B[i],Bs[k],N)/Norm(Bs[k],N);
   for(j=0;j<N;j++){
    Bs[i][j]-=mu[i][k]*Bs[k][j];
   }
  }
 }
 return Bs;
}
```

Code 2. The inner product module.

```
long long int inner_product(long long int* v,long long int* w,int N)
{
 long long int result=0;
 for(int i=0;i<N;i++){
  result+=v[i]*w[i];
 }
 return result;
}
```

The modular software library is developed by following the 64-bit architecture to perform operations without any overflow problem. The modular software infrastructure library's source codes and efficient implementations are available at https://github.-com/hsatilmis/modular-_software_library (accessed on 2 July 2021).

Section 3.1 provides the software features and structures of the efficient implementations of GaussSieve, ProGaussSieve, ENUM, and BKZ developed using the software library.

3.1. The GaussSieve and the ProGaussSieve Implementations

The efficient implementations of the GaussSieve and the ProGaussSieve algorithms, which are developed by using the modular software library as an infrastructure, are constructed by using the C programming language. In the developed implementations, variables *pointer* or *double* − *pointer* are used frequently to minimize the delay to access the data in the memory. The data structures *struct* provided by the C programming language are preferred in these implementations for the variables that contain many values, such as lattice. Since the software library is used as an infrastructure for the efficient implementations, they are developed under the 64-bit architecture to be compatible with this library. Therefore, the integer variables are defined in the data type *long long int*. Code 3 shows the data structure *struct*, which defines the array variable (Coord) where the coordinate values of the basis vectors of the lattice consisting of the integer vectors are stored, and the variable of the Euclidean norms (Norm2) of these vectors.

Code 3. The data structure for the basis vectors.

```
struct vect{
 long long int Coord[N];
 unsigned long long int Norm2;
};
```

The GaussSieve and the ProGaussSieve algorithms frequently compute the vectorial mathematical operations for each iteration. For this reason, to have efficient implementations, mathematical operation modules in the modular software infrastructure library are used. The Gram–Schmidt module in the software library is used in the efficient implementations to obtain the reduced lattice that the other implementations receive as input parameters. In Observations 1 and 2, more details are given on the termination criterion and the number of collisions. These are helpful to understand the main idea of the performance improvements.

Observation 1. *In the GaussSieve implementation developed by Micciancio and Voulgaris in [5], the lattices given as input parameters are randomly generated. In this implementation, it is difficult to estimate the total number of collisions (the termination criterion) where the shortest vectors of the randomly generated lattices are found. Furthermore, the shortest vectors of the randomly generated lattices are unknown, and the accuracy of the shortest vector found by the implementation cannot be theoretically proven.*

Observation 2. *It is not easy to determine the total number of collisions, which is the termination criterion of sieving algorithms [41]. In other words, when a small value is chosen as the termination criterion, the implementations of sieving algorithms may stop working before they find the shortest lattice vector. On the other hand, when an enormous value is chosen as the termination criterion,*

implementations can continue their work even after finding the shortest lattice vector. As a result, the implementations can cause unnecessary resource usage.

In Remark 1, the comparison details are given.

Remark 1. *To make a logical comparison of the efficient implementations of GaussSieve and ProGaussSieve with the GaussSieve and the ProGaussSieve implementations in [6] about the run time complexities, the memory space is used as a termination criterion in this paper. On the other hand, the use of memory space as a termination criterion provides a different perspective to the solution of the termination criterion determination problem in sieving algorithms. The memory space values that implementations in the literature expend upon when they find the shortest vectors are set as the termination criterion for the efficient implementations of GaussSieve and ProGaussSieve developed in this paper.*

The lattices generated by using the SageMath application are given as the input parameters to the efficient implementations in this paper. The efficient implementations return the shortest vectors as output when consuming the memory spaces are selected as a termination criterion. Both the efficient implementations in this paper and the implementations in the literature give the shortest vectors of the input lattices as output within the same memory spaces.

3.2. The HashSieve Implementation

The efficient implementation of the HashSieve algorithm was developed based on the HashSieve implementation in [10]. Unlike the HashSieve implementation in [10], the efficient implementation of HashSieve, which uses the modular software infrastructure library, was built by using the C programming language and by following the 64-bit architecture.

While the efficient implementation of HashSieve was developed, the variables were defined in structures *pointer* and $double - pointer$ to reduce the run time of an implementation. In an efficient implementation, data structures *struct* are used for variables such as lattices with different properties. In addition to the data structures *struct* in the HashSieve implementation in [10], the structures *struct* shown in Code 4 are used. The variables in the data structures *struct* in Code 4 are defined in structures *pointer* or $double - pointer$ to provide faster access to data in the memory. On the other hand, in the HashSieve implementation in [10], the same variables are defined as standard arrays. This difference in the defining variables is one of the factors that allow the efficient implementation of HashSieve to perform better than the HashSieve implementation in [10] regarding the run time. The data structures *struct* in Code 4 define the coordinates (matrix, dmatrix) and the Euclidean norms (matrixNorm, dmatrixNorm) of the data type *long long int* and *double* lattices. As input parameters for the efficient implementation of HashSieve, the lattices used in the HashSieve implementation in [10] are given. The efficient implementation of HashSieve outputs the same shortest lattice vectors as the HashSieve implementation in [10] for the same lattice samples.

Code 4. The different data structures in the efficient implementation of HashSieve.

```
struct matrix{
 long long int** B;
};

struct matrixNorm{
 unsigned long long int* Norm2;
};

struct dmatrix{
 double** Bs;
};

struct dmatrixNorm{
 double* Norm2;
};
```

3.3. The ENUM and The BKZ Implementations

The ENUM algorithm is used as a subprocess to find the smallest vector. This corresponds to obtaining the new lattice vector in the BKZ reduction algorithm of Schnorr and Euchner. While implementing the ENUM, the variables are mostly defined in the structures *pointer* or *double* − *pointer* so that the implementation can have a low run time. Since the ENUM implementation uses the software library with the 64-bit architecture as the infrastructure, the integer variables are defined in the *long long int* data type in this implementation. For the vector arithmetic, the arithmetic operations modules in the software library are used. The developed ENUM implementation is added as a module to the modular software library for Schnorr and Euchner's efficient implementation of BKZ.

The efficient and practical version of Schnorr and Euchner's BKZ algorithm is implemented using the modular software library developed in this paper as the infrastructure. The module of the LLLFP reduction algorithm in the software library used by the BKZ algorithm as a subprocess is used in the efficient implementation of BKZ. In Observation 3, the zero vector problem in LLLFP algorithm is defined.

Observation 3. *While testing the module in the software library of the LLLFP algorithm, which is developed based on the LLL reduction algorithm and designed to minimize floating-point errors, the zero vector problem is encountered. Given some particular lattice examples as input parameters, the LLLFP module is calculated as the first lattice vector to be a zero vector during the reduction process. It is not desired situation in the LLLFP algorithm for the first vector in the lattices to be a zero vector, as the reduction operation cannot proceed correctly due to the zero vector problem.*

In Remark 2, the solution for the zero vector problem in LLLFP algorithm is defined.

Remark 2. *In [8], to solve the zero vector problem in the LLLFP reduction algorithm, it is proposed to remove the zero vector from the lattice and to continue the reduction without the zero vector. This proposed solution is implemented in the LLLFP module in the software library with a slight difference.*

The module of the ENUM algorithm in the software library is used as a subprocess in the efficient implementation of BKZ and to find the smallest element. The Gram–Schmidt module in the modular software infrastructure library is used to calculate the Gram–Schmidt constants and the Euclidean norms of Gram–Schmidt reduced lattice vectors used in the efficient implementation of BKZ developed using the C programming language. Since the vectorial arithmetic operations are carried out continuously in the BKZ algorithm, the arithmetic operations module in the software library is developed in the BKZ implementation. In the BKZ implementation developed in accordance with the 64-bit architecture, the integer variables are defined in the *long long int* data type. In addition,

the data structures *struct* and the structures *pointer* or *double* − *pointer* are commonly used in the efficient implementation of BKZ.

4. Experimental Results

In this section, we give the details about the experimental results.

4.1. Settings

The efficient implementations and the modular software library were developed and tested on the x86 solution platform in Visual Studio Community 2017 version 15.9.11. The compiler's default compilation options were selected when running the library. The average run times of the efficient implementations developed in this paper were measured on a server computer with 2× Intel Xeon E5-2630V4 (20 Core) processors and 64 GB of RAM hardware.

4.2. Results

By running each efficient implementation at least 1000 times, the average run times of the implementation were calculated. In Table 2, the average run times are given. Since the real run times are used in Table 2, the average run times of the implementations in the literature are not included in this table for comparison.

Table 2. The comparison of the run times of the efficient implementations (*n*: lattice size, MS: memory space, and RT: run time).

Efficient Implementations	*n*	MS (mb)	RT (sec)
GaussSieve	30	0.450264	0.124030
	35	0.890092	0.227235
	40	1.780020	0.391660
	45	3.565036	0.808455
	50	7.130300	1.698730
	55	14.260128	3.817835
	60	28.505092	8.242560
	65	57.020392	17.794480
	70	114.050484	37.870260
ProGaussSieve	30	0.450264	0.074120
	35	0.890092	0.127470
	40	1.780020	0.184480
	45	3.565036	0.305690
	50	7.130300	0.570365
	55	14.260128	1.038980
	60	28.505092	1.976725
	65	57.020392	3.819100
	70	114.050484	7.347520
HashSieve v	30	0.008760	0.004100
	35	0.185558	0.242220
	40	0.438350	0.978350
	45	0.927556	4.487940
	50	2.145861	24.607380
	55	4.771455	122.636160
	60	10.554869	558.496300
	65	24.262557	3055.551765
	70	54.949216	16,309.201660
BKZ	30	-	0.005524
	35	-	0.009241
	40	-	0.014419
	45	-	0.021529
	50	-	0.031721
	55	-	0.046587
	60	-	0.077366
	65	-	0.080678
	70	-	0.104639

The lattices randomly generated by the SageMath application were used as input parameters to test the efficient implementations of GaussSieve and ProGaussSieve. When the outputs of the efficient implementations of GaussSieve and ProGaussSieve were evaluated, it is observed that implementations find the shorter vectors in each iteration and return the shortest vector possible. By considering the experimental results, the run time complexities are computed in a big-O notation using real run times. The linear functions and the curves representing the exponential values of the run time complexities of the efficient implementations by the lattice sizes are given in Figure 2.

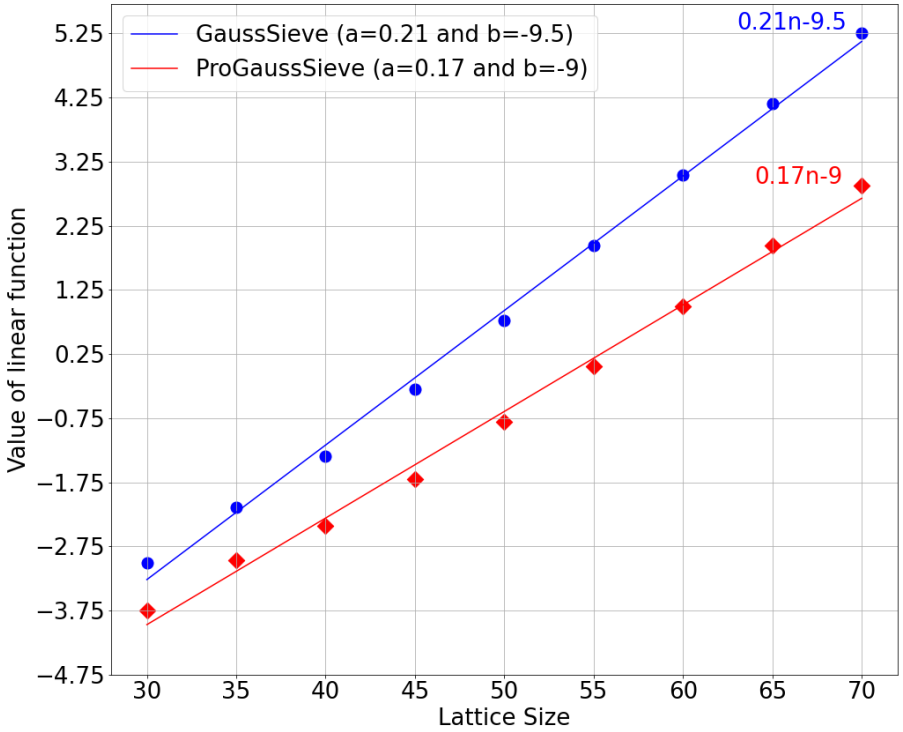

Figure 2. The exponential values of the run time complexities.

In Remark 3, the computation details of the complexity of the GaussSieve and ProGaussSieve are discussed. Note that the complexities are computed with the experimental results.

Remark 3. *The run time complexities of the efficient implementations of GaussSieve and ProGaussSieve were computed using the curve fitting method using the Linear Regression model [42]. The exponential run time complexities were calculated using the real run times in Table 2. The exponential values are assumed as a linear function ($y = an + b$). In this context, the linear function's approximate values of the constants a and b are estimated by forming the Linear Regression model. By estimating the values of the constants a and b, the run time complexities of the efficient implementations were also obtained (considering $2^y = 2^{an+b}$). As a result, the run time complexities of the implementations of GaussSieve and ProGaussSieve were found to be $2^{0.21n-9.5}$ and $2^{0.17n-9}$, respectively. The run time complexities of the efficient implementations and the implementations in [6] are given in Table 3. The source codes of the Linear Regression model, which was developed using the scikit-learn module [43] on the Python programming language, are available at https://github.com/hsatilmis/modular_software_library/-blob/master/(pro)gausssieve_curve_fitt-ing.ipynb (accessed on 2 July 2021).*

Table 3. The comparison of the run time complexities of the implementations.

Algorithms	Implementations	Run Time Complexity
GaussSieve	This Paper	$2^{0.21n-9.5}$
	[6]	$2^{0.52n-22}$
ProGaussSieve	This Paper	$2^{0.17n-9}$
	[6]	$2^{0.49n-25}$

In Remark 4, the performance improvements are discussed.

Remark 4. *In the efficient implementations of GaussSieve and ProGaussSieve, the space complexity values obtained from the results in [6] were used as the termination criterion. Therefore, the efficient implementations of GaussSieve and ProGaussSieve and the implementations in [6] use the same memory spaces. According to the experimental results with the small lattice sizes to compare the run times, the developed implementation for GaussSieve is at least 70% faster than the GaussSieve implementation in [6]. Moreover, the run time of ProGaussSieve is improved by almost 75% compared to the Laarhoven and Mariano's ProGaussSieve implementations in [6].*

The lattices used in the HashSieve implementation in [10] are given to the efficient implementation as input parameters to test the efficient implementation of HashSieve. The outputs of this efficient implementation were compared with those of HashSieve implementation, and it is concluded that the efficient implementation works correctly. In Remark 5, the performance analysis of HashSieve implementation is discussed.

Remark 5. *Considering the experimental results for all lattice sizes, the proposed HashSieve implementation is at least 49% more efficient than Laarhoven's standard HashSieve implementation in [10].*

The randomly generated lattices using the SageMath application are used as input parameters to test the efficient implementation of BKZ. Moreover, the same lattice samples are given as input parameters to the BKZ algorithm in the SageMath. The outputs of the BKZ algorithm in the SageMath application were compared with those of the efficient implementation of BKZ. As a result of the comparison, except for some unique lattice samples given as input, it is observed that the efficient implementation of BKZ gives correct outputs. In Remark 6, a discussion on the efficiency of run times is given.

Remark 6. *There are two main reasons why the efficient implementations of algorithms developed in this paper have better run times than previous ones.*

1. *The common subcomponents in the algorithms are used as subprocesses. The algorithms constantly need common subcomponents during their operations. Hence, the run time of the implementations is directly affected by the common subcomponents. For this reason, a modular software library is developed that includes the common subcomponents as modules. During modular software library development, often, the structures pointer or double − pointer are used effectively in the variable definitions in the modules. Due to the pointer structures that provide quick access to the data in the memory, the processing speed of the modules is increased. Therefore, the run time of efficient implementations, which use the modules as subprocesses, are improved.*
2. *In this structure, the vector arithmetic in the lattice is needed. Therefore, the data structures are needed to define the vectors and the lattice structures in the efficient implementations. To obtain efficient implementation, the data structures pointer and struct were used. This helps to quickly access the data and the vector elements.*

5. Conclusions and Future Works

In this paper, a modular software infrastructure library was developed to provide an infrastructure for efficient implementations of the sieving, enumeration, and reduction algorithms. Using the modules in this software library, efficient implementations of the GaussSieve, ProGaussSieve, HashSieve, ENUM, and BKZ algorithms were developed. The outputs of the efficient implementations developed were compared with those of the implementations in the literature. Moreover, the correctness of the developed implementation was assessed by comparing the outputs with the previous ones. The run times and the memory usage of these efficient implementations were provided. The run time complexities of the efficient implementations of GaussSieve and ProGaussSieve were calculated and compared with examples in the literature. It is concluded that the efficient implementations of GaussSieve and ProGaussSieve, which use the memory space as a termination criterion, have better run time complexities than the implementations in the literature. According to the experimental results, the efficient implementations of GaussSieve and ProGaussSieve are at least 70% and 75% more efficient in terms of run time than previous ones, respectively. Finally, the efficient implementation of HashSieve is at least 49% more efficient in terms of the run time than the sample in the literature that is used as its basis during the development. In future studies, we aim to develop parallel versions of the implementations developed in this paper in order to be more efficient.

Author Contributions: Conceptualization, methodology, and investigation: H.S. and S.A., writing—original draft preparation and writing—review and editing: H.S., S.A. and C.-C.L. All authors have read and agreed to the published version of the manuscript.

Funding: H. Satılmış and S. Akleylek were partially supported by TÜBİTAK under grant no.EEEAG-117E636.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The date can be found at https://github.com/hsatilmis/modular_software_library (accessed on 2 July 2021). https://github.com/hsatilmis/modular_software_library/blob/master/(pro)gausssieve_curve_fitting.ipynb (accessed on 2 July 2021).

Acknowledgments: We the associate editor and the anonymous reviewers for their suggestions, which improved not only the readability and organization but also the quality of the paper.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Bernstein, D.J. Introduction to post-quantum cryptography. In *Post-Quantum Cryptography*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–14.
2. Shor, P.W. Algorithms for quantum computation: discrete logarithms and factoring. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 20–22 November 199 ; IEEE: Piscataway, NJ, USA, 1994; pp. 124–134.
3. Aghili, S.F.; Mala, H. Security analysis of an ultra-lightweight RFID authentication protocol for m-commerce. *Int. J. Commun. Syst.* **2019**, *32*, e3837. [CrossRef]
4. Aghili, S.F.; Mala, H.; Schindelhauer, C.; Shojafar, M.; Tafazolli, R. Closed-loop and open-loop authentication protocols for blockchain-based IoT systems. *Inf. Process. Manag.* **2021**, *58*, 102568. [CrossRef]
5. Micciancio, D.; Voulgaris, P. Faster exponential time algorithms for the shortest vector problem. In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, USA, 17–19 January 2010; SIAM: Philadelphia, PA, USA, 2010; pp. 1468–1480.
6. Laarhoven, T.; Mariano, A. Progressive lattice sieving. In Proceedings of the International Conference on Post-Quantum Cryptography, Fort Lauderdale, FL, USA, 9–11 April 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 292–311.
7. Schnorr, C.P.; Euchner, M. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In Proceedings of the International Symposium on Fundamentals of Computation Theory, Gosen, Germany, 9–13 September 1991; Springer: Berlin/Heidelberg, Germany, 1991; pp. 68–85.

8. Schnorr, C.; Euchner, M. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.* **1994**, *66*, 181–199. [CrossRef]
9. Laarhoven, T.; de Weger, B. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In Proceedings of the International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, 23–26 August 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 101–118.
10. Laarhoven, T. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Proceedings of the Annual Cryptology Conference, Santa Barbara, CA, USA, 16–20 August 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 3–22.
11. Ajtai, M.; Kumar, R.; Sivakumar, D. A sieve algorithm for the shortest lattice vector problem. In Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, Hersonissos, Greece, 6–8 July 2001; ACM: New York, NY, USA, 2001; pp. 601–610.
12. Nguyen, P.Q.; Vidick, T. Sieve algorithms for the shortest vector problem are practical. *J. Math. Cryptol.* **2008**, *2*, 181–207. [CrossRef]
13. Charikar, M.S. Similarity estimation techniques from rounding algorithms. In Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, Montreal, QC, Canada, 19–21 May 2002; ACM: New York, NY, USA, 2002; pp. 380–388.
14. Qi, J.Y.; Gang, H.H. Using K-Means LSH to Speed up Solving the Shortest Vector Problem. *J. Cryptogr. Res.* **2020**, *7*, 473. [CrossRef]
15. Shoup, V.; others. NTL: A Library for Doing Number Theory. 2001. Available online: http://www.shoup.net/ntl (accessed on 2 July 2021).
16. Milde, B.; Schneider, M. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In Proceedings of the International Conference on Parallel Computing Technologies, Kazan, Russia, 19–23 September 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 452–458.
17. Ishiguro, T.; Kiyomoto, S.; Miyake, Y.; Takagi, T. Parallel Gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In Proceedings of the International Workshop on Public Key Cryptography, Buenos Aires, Argentina, 26–28 March 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 411–428.
18. Mariano, A.; Timnat, S.; Bischof, C. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. In Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, Paris, France, 22–24 October 2014; IEEE: Piscataway, NJ, USA 2014; pp. 278–285.
19. Yang, S.Y.; Kuo, P.C.; Yang, B.Y.; Cheng, C.M. Gauss sieve algorithm on GPUs. In Proceedings of the Cryptographers' Track at the RSA Conference, San Francisco, CA, USA, 14–17 February 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 39–57.
20. Falcao, G.; Cabeleira, F.; Mariano, A.; Paulo Santos, L. Heterogeneous Implementation of a Voronoi Cell-Based SVP Solver. *IEEE Access* **2019**, *7*, 127012–127023. [CrossRef]
21. Kannan, R. Improved algorithms for integer programming and related lattice problems. In Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, Boston, MA, USA, 25–27 April 1983; ACM: New York, NY, USA, 1983; pp. 193–206.
22. Fincke, U.; Pohst, M. A procedure for determining algebraic integers of given norm. In Proceedings of the European Conference on Computer Algebra, London, UK, 28–30 March 1983; Springer: Berlin/Heidelberg, Germany, 1983; pp. 194–202.
23. Gama, N.; Nguyen, P.Q.; Regev, O. Lattice enumeration using extreme pruning. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, France, 30 May–3 June 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 257–278.
24. Dagdelen, Ö.; Schneider, M. Parallel enumeration of shortest lattice vectors. In Proceedings of the European Conference on Parallel Processing, Ischia, Italy, 31 August–3 September 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 211–222.
25. Correia, F.; Mariano, A.; Proenca, A.; Bischof, C.; Agrell, E. Parallel improved Schnorr-Euchner enumeration SE++ for the CVP and SVP. In Proceedings of the 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Greece, 17–19 February 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 596–603.
26. Nguyen, P.Q.; Vallée, B. *The LLL Algorithm*; Springer: Berlin/Heidelberg, Germany, 2010.
27. Gama, N.; Nguyen, P.Q. Predicting lattice reduction. In Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, 13–17 April 2008; Springer: Berlin/Heidelberg, Germany, 2008; pp. 31–51.
28. McGuire, G.; Robinson, O. Lattice Sieving in Three Dimensions for Discrete Log in Medium Characteristic. *J. Math. Cryptol.* **2020**, *15*, 223–236. [CrossRef]
29. Hanrot, G.; Pujol, X.; Stehlé, D. Analyzing blockwise lattice algorithms using dynamical systems. In Proceedings of the Annual Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 447–464.
30. Schnorr, C.P. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.* **1987**, *53*, 201–224. [CrossRef]
31. Chen, Y.; Nguyen, P.Q. BKZ 2.0: Better lattice security estimates. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Seoul, Korea, 4–8 December 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 1–20.
32. Liu, X.; Fang, X.; Wang, Z.; Xie, X. A new parallel lattice reduction algorithm for BKZ reduced bases. *Sci. China Inf. Sci.* **2014**, *57*, 1–10. [CrossRef]
33. Correia, F.J.G. Assessing the Hardness of SVP Algorithms in the Presence of CPUs and GPUs. Ph.D. Thesis, Minho University, Braga, Portugal, 2014.

34. Mariano, A.; Laarhoven, T.; Correia, F.; Rodrigues, M.; Falcao, G. A Practical View of The State-of-The-Art of Lattice-Based Cryptanalysis. *IEEE Access* **2017**, *5*, 24184–24202. [CrossRef]
35. The Sage Developers. SageMath, the Sage Mathematics Software System (Version 8.6). 2019. Available online: https://www.sagemath.org (accessed on 2 July 2021).
36. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM (JACM)* **2009**, *56*, 1–40. [CrossRef]
37. Laarhoven, T.; van de Pol, J.; de Weger, B. Solving Hard Lattice Problems and the Security of Lattice-Based Cryptosystems. *IACR Cryptol. EPrint Arch.* **2012**, *2012*, 533.
38. Klein, P. Finding the closest lattice vector when it's unusually close. In Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, 9–11 January 2000; SIAM: Philadelphia, PA, USA, 2000; pp. 937–941.
39. Raghavan, P.; Tompson, C.D. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica* **1987**, *7*, 365–374. [CrossRef]
40. Mariano, A.; Bischof, C.; Laarhoven, T. Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP. In Proceedings of the 2015 44th International Conference on Parallel Processing, Beijing, China, 1–4 September 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 590–599.
41. Plantard, T.; Schneider, M. Creating a Challenge for Ideal Lattices. *IACR Cryptol. EPrint Arch.* **2013**, *2013*, 39.
42. Montgomery, D.C.; Peck, E.A.; Vining, G.G. *Introduction to Linear Regression Analysis*; John Wiley & Sons: Hoboken, NJ, USA, 2021.
43. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.