Tuning GaussSieve for Speed

Robert Fitzpatrick[1], Christian Bischof[2], Johannes Buchmann[3], Özgür Dagdelen[3], Florian Göpfert[3], Artur Mariano[2] and Bo-Yin Yang[1]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
[2] Institute for Scientific Computing TU Darmstadt, Darmstadt, Germany
[3] CASED, TU Darmstadt, Darmstadt, Germany

Abstract. The area of lattice-based cryptography is growing ever-more prominent as a paradigm for quantum-resistant cryptography. One of the most important hard problem underpinning the security of lattice-based cryptosystems is the *shortest vector problem* (SVP). At present, two approaches dominate methods for solving instances of this problem in practice: enumeration and sieving. In 2010, Micciancio and Voulgaris presented a heuristic member of the sieving family, known as GaussSieve, demonstrating it to be comparable to enumeration methods in practice. With contemporary lattice-based cryptographic proposals relying largely on the hardness of solving the shortest and closest vector problems in *ideal* lattices, examining possible improvements to sieving algorithms becomes highly pertinent since, at present, only sieving algorithms have been successfully adapted to solve such instances more efficiently than in the random lattice case. In this paper, we propose a number of heuristic improvements to GaussSieve, which can also be applied to other sieving algorithms for SVP.

1 Introduction

Lattice-based cryptography is gaining increasing traction and popularity as a basis for post-quantum cryptography, with the Shortest Vector Problem (SVP) being one of the most important computational problems on lattices. Its difficulty is closely related to the security of most lattice-based cryptographic constructions to date. The SVP consists in finding a shortest (with respect to a particular, usually Euclidean, norm) non-zero lattice point in a given lattice.

For solving SVP instances, we have a choice of algorithms available. In recent works, heuristic variants of Kannan's simple enumeration algorithm have dominated. The original algorithm [11] solves SVP (deterministically) with time complexity $n^{\frac{n}{2}+o(n)}$ (n being the lattice dimension). More recent works (such as [7]) allow probabilistic SVP solution, sacrificing guaranteed solution for runtime improvements.

A more recently-studied family of algorithms is known as lattice sieving algorithms, introduced in the 2001 work of Ajtai et al. [4]. In 2008, Nguyen and Vidick [20] presented a careful analysis of the algorithm of Ajtai et al., showing it to possess time complexity of $2^{5.90n+o(n)}$ and space complexity $2^{2.95n+o(n)}$.

Heuristic variants of [20], which run significantly faster than proven lower bounds are presented in [20,26,27]. In 2010, Micciancio and Voulgaris [18] proposed two new algorithms: ListSieve and a heuristic derivation known as GaussSieve, with GaussSieve being the most practical sieving algorithm known at present. While no runtime bound is known for GaussSieve, the use of a simple heuristic stopping condition, in practice, appears effective with no cases being known (to the best of our knowledge) in which GaussSieve fails to return a shortest non-zero vector.

For purposes of enhanced communication, computation and memory complexity, many recent lattice-based cryptographic proposals employ *ideal* lattices rather than "random" lattices. Ideal lattices, in brief, possess significant additional structure which allows much more attractive implementation of said proposals. However, as with any introduction of structure, the question of any simultaneously-introduced weakening of the underlying problems arises. In 2011, Schneider [21] illustrated that (following a suggestion in the work of Micciancio and Voulgaris [18]) one can take advantage of the additional structure present in ideal lattices in a simple way to obtain substantial speedups for such cases. Interestingly, no such comparable techniques are known for other SVP algorithms, with only sieving algorithms appearing to be capable of exploiting the additional structure exposed in ideal lattices.

Another attractive feature of sieving algorithms is their relative amenability to parallelization. Also in 2011, Milde and Schneider [19] proposed a parallel implementation of GaussSieve, though the methodology used limited the number of threads to about ten, before no substantial further speedups could be obtained. In 2013, Ishiguro et al. [10] proposed a somewhat more natural parallelization of GaussSieve, allowing a much larger number of threads. Using such an approach, they report the solution of the 128-dimensional ideal lattice challenge [2] in 30,000 CPU hours. Currently, the most efficient GaussSieve implementation (of which details have been published) is due to Mariano et al [16] who implemented GaussSieve with a particular effort to avoid resource contention. In this work, we exhibit several further speedups which can be obtained both in the random and ideal lattice cases.

While the security of most lattice-based cryptographic constructions relies on the difficulty of *approximate* versions of the related Closest Vector Problem (CVP) and SVP, the importance of improving *exact* SVP solvers stems from their use (following Schnorr's hierarchy [22]) in the construction of approximate CVP/SVP solvers. Thus, any improvements, both theoretically and experimentally, in exact SVP solvers can lead to a need for re-appraisal of proposed parameterizations.

Our Contribution. In this work, we highlight several practical improvements that are applicable to other sieving algorithms. In particular, we propose the following optimizations, which we incorporated into GaussSieve:

- We correct an error in the Gaussian sampler of the reference implementation of Voulgaris and propose an *optimized Gaussian sampler* in which we dynamically adapt the Gaussian parameter used during the execution of the

algorithm. Our experiments show that GaussSieve with our optimized Gaussian sampler requires significantly fewer iterations to terminate and leads to a speedup of up to 3.0× over the corrected reference implementation in random lattices in dimension 60-70.
- The use of *multiple randomized bases* to seed the list before running the sieving process offers substantial efficiency gains. Indeed, the speedup appears to grow linearly in the dimension of the underlying lattice.
- We introduce a very efficient heuristic to compute *a first approximation to the angle between two vectors* in order to test cheaply whether there is the need to compute full inner products for the reduction process. This optimization is possibly of independent interest beyond sieving algorithms.

We note that our improvements can be integrated into parallel versions of GaussSieve without complication or restriction.

2 Background and Notation

A (full-rank) lattice $\varLambda$ in $\mathbb{R}^n$ is a discrete additive subgroup. For a general introduction, the reader is referred to [17]. We view a lattice as being generated by a (non-unique) basis $\mathbf{B} = \{\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}\} \subset \mathbb{R}^n$ of linearly-independent vectors. We assume that the vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}$ form the rows of the $n \times n$ matrix $\mathbf{B}$. That is,

$$\varLambda = \mathcal{L}(\mathbf{B}) = \mathbb{Z}^n \cdot \mathbf{B} = \left\{ \sum_{i=0}^{n-1} x_i \cdot \mathbf{b}_i \mid x_0, \ldots, x_{n-1} \in \mathbb{Z} \right\} .$$

The rank of a lattice $\varLambda$ is the dimension of the linear span $\mathrm{span}(\varLambda)$ of $\varLambda$. The basis $\mathbf{B}$ is not unique, and thus we call two bases $\mathbf{B}$ and $\mathbf{B}'$ *equivalent* if and only if $\mathbf{B}' = \mathbf{B}\mathbf{U}$ where $\mathbf{U}$ is a unimodular matrix, i.e., an integer matrix with $|\det(\mathbf{U})| = 1$. We note that such unimodular matrices form the general linear group $GL_n(\mathbb{Z})$. Being a discrete subgroup, in any lattice there exists a subset of vectors which possess minimal (non-zero) norm amongst all vectors. When asked to solve the shortest vector problem, we are given a lattice basis and asked to deliver a member of this subset. SVP is known to be NP-hard under randomized reductions [3].

Random Lattices. Throughout this work, we rely on experiments with "random" lattices. However, the question of what a "random" lattice is and how to generate a random basis of one are non-trivial. In a mathematical sense, an answer to the definition of a random lattice follows from a work in 1945 by Siegel [24], with efficient methods for sampling such random lattices being proposed, for instance, by Goldstein and Mayer [9]. In this work, all experiments were conducted with Goldstein-Mayer lattices, as provided by the TU Darmstadt Lattice Challenge project. For more details, the reader is directed to [8].

Definition 1. *Given two vectors* $\mathbf{v}, \mathbf{w}$ *in a lattice* $\varLambda$*, we say that* $\mathbf{v}, \mathbf{w}$ *are Gauss-reduced if*

$$\min(\|\mathbf{v} \pm \mathbf{w}\|) \geq \max(\|\mathbf{v}\|, \|\mathbf{w}\|) .$$

Lattice Basis Reduction. A given lattice has an infinite number of bases. The aim of lattice basis reduction is to transform a given lattice basis into one which contains vectors which are both relatively short and relatively orthogonal. Such bases, in some sense, allow easier and/or more accurate solutions of approximation variants of SVP or its related problem, the Closest Vector Problem (CVP). In practice, the most effective arbitrary-dimension lattice basis reduction algorithms are descendants of the LLL algorithm [14], with the Block-Korkine-Zolotarev (BKZ) family [22,5] (or framework) of algorithms being the most effective in practice. The LLL and BKZ algorithms rely on successive exact SVP solution in a number of projected lattices. These projected lattices are two-dimensional in the case of LLL and of arbitrary dimension in the case of BKZ – the (maximal) projected lattice dimension being termed the "blocksize" in BKZ. For more details, the reader is referred to [6].

Balls and Spheres. We define the Euclidean n-sphere $\mathcal{S}_n(\mathbf{x}, r)$ centered at $\mathbf{x} \in \mathbb{R}^{n+1}$ and of radius r by $\mathcal{S}_n(\mathbf{x}, r) := \{\mathbf{y} \in \mathbb{R}^{n+1} \colon \parallel \mathbf{x} - \mathbf{y} \parallel = r\}$. The (open) Euclidean n-ball $\mathcal{B}_n(\mathbf{x}, r)$ centered at $\mathbf{x} \in \mathbb{R}^n$ and of radius r is defined to be $\mathcal{B}_n(\mathbf{x}, r) := \{\mathbf{y} \in \mathbb{R}^n \colon \parallel \mathbf{x} - \mathbf{y} \parallel < r\}$.

Gaussians. The discrete Gaussian distribution with parameter s over a lattice Λ is defined to be the probability distribution with support Λ which, for each $\mathbf{x} \in \Lambda$, assigns probability proportional to $\exp(-\pi\|\mathbf{x}\|^2/s^2)$.

Miscellany. We use $\oplus$ to denote the bitwise XOR operation and use $\mathbf{a}\angle\mathbf{b}$ to denote the angle between vectors $\mathbf{a}$ and $\mathbf{b}$. Given a binary vector $\mathbf{a}$, we use $\mathrm{w}(\mathbf{a})$ to denote the Hamming weight of $\mathbf{a}$.

3 The GaussSieve Algorithm

In 2010, Micciancio and Voulgaris [18] introduced the GaussSieve algorithm. GaussSieve is a heuristic efficient variant of the ListSieve algorithm. In contrast to GaussSieve, for ListSieve there exist provable bounds on the running time and space requirements. Algorithm 1 depicts the GaussSieve algorithm in more detail.

GaussSieve operates upon a supplied lattice basis $\mathbf{B}$. It utilizes a dynamic list L of lattice points. At each iteration, GaussSieve samples a new lattice point – typically with Klein's algorithm [12] – and attempts to reduce that vector against vectors in the list L. By "reducing" we mean adding an integer multiple of a list vector such that the norm of the resulting vector is reduced. Once the vector cannot be reduced further by list members, the resulting vector is incorporated in the list. Afterwards, all the vectors in the list L are tested to determine if they can be reduced against this new vector. If so, those vectors are removed to a stack S, with the stack playing the role of Klein's algorithm in subsequent iterations till it is depleted. This ensures that all vectors in the list L remain pairwise Gauss-reduced at any point during the execution of the algorithm. Eventually,

Algorithm 1: GaussSieve

```
Input   : Basis $\mathbf{B}$, collision limit $c$
Output: $\mathbf{v} : \mathbf{v} \in \Lambda(\mathbf{B}) \wedge \|\mathbf{v}\| = \lambda_1(\mathbf{B})$
$\mathsf{L} \leftarrow \{\}, \mathsf{S} \leftarrow \{\}, \mathsf{col} \leftarrow 0$
while $\mathsf{col} < c$ do
    if $\mathsf{S}$ is not empty then
        $\mathbf{v} \leftarrow \mathsf{S}$.pop()
    else
        $\mathbf{v} \leftarrow$ SampleKlein($\mathbf{B}$)
    j $\leftarrow$ GaussReduce($\mathbf{v}$,$\mathsf{L}$, $\mathsf{S}$)
    if $j$ = true then
        $\mathsf{col} \leftarrow \mathsf{col} + 1$
return $\mathbf{v} \in \mathsf{L}$ s.t. $\|v\| = \min_{\mathbf{x} \in \mathsf{L}} \|\mathbf{x}\|$
```

```
function GaussReduce($\mathbf{p}$,$\mathsf{L}$,$\mathsf{S}$)
was_reduced $\leftarrow$ true
while was_reduced = $true$ do
    was_reduced $\leftarrow$ false
    for all $\mathbf{v}_i \in \mathsf{L}$ do
        if $\exists t \in \mathbb{Z}$: $\|\mathbf{p} + t\mathbf{v}_i\| < \|\mathbf{p}\|$ then
            $\mathbf{p} \leftarrow \mathbf{p} + t\mathbf{v}_i$
            was_reduced $\leftarrow$ true
if $\|\mathbf{p}\| = 0$ then
    return true
for all $\mathbf{v}_i \in \mathsf{L}$ do
    if $\exists u \in \mathbb{Z}$: $\|\mathbf{v}_i + u\mathbf{p}\| < \|\mathbf{v}_i\|$ then
        $\mathsf{L} \leftarrow \mathsf{L} \backslash \{\mathbf{v}_i\}$
        $\mathbf{v}_i \leftarrow \mathbf{v}_i + u\mathbf{p}$
        $\mathsf{S}$.push($\mathbf{v}_i$)
$\mathsf{L} \leftarrow \mathsf{L} \cup \{\mathbf{p}\}$
return false
end function
```

by this iterative process, the shortest vector in the lattice is found (with high probability). In the following, we detail the GaussSieve algorithm in several aspects.

Sampling. In order to populate the list with reasonably short vectors, GaussSieve samples lattice points via Klein's randomized algorithm [12], following the suggestion in [20]. Klein's algorithm, upon input a lattice basis $\mathbf{B}$ outputs a lattice point distributed according to a zero-centered Gaussian of parameter s over the lattice $\Lambda(\mathbf{B})$. Since the vectors so derived are small integer combinations of the supplied basis vectors, the norms of these vectors are strongly dependent on the "quality" of the supplied basis. Hence, reducing the input basis with "stronger" lattice-reduction algorithms yields shorter vectors output and thus, intuitively and in practice, GaussSieve terminates earlier than when given a "less-reduced" basis from which to sample vectors. However, while the cost of enumeration algorithms is strongly affected by the strength of the lattice-reduction employed, such a strong correspondence does not appear to hold for the case of GaussSieve - such issues are discussed further in Section 5.

Reduction. We attempt to reduce the given vector $\mathbf{p}$ (obtained either from Klein's algorithm or from the stack) against all list vectors, i.e., we try to find a list vector $\mathbf{v}$ and integer t such that $\|\mathbf{p} + t\mathbf{v}\| < \|\mathbf{p}\|$, in which case we reduce $\mathbf{p}$ using $\mathbf{v}$. Once no such $\mathbf{v}$ exists in the list, we attempt to reduce the extant list vectors against $\mathbf{p}$. All list vectors which can be reduced using $\mathbf{p}$ are duly reduced, removed from the list and inserted to the stack S. As a result, GaussSieve maintains its list L in a pairwise reduced state at the close of every iteration.

In the following iteration, if the stack contains at least one element, we pop a vector from the stack in lieu of employing Klein's algorithm.

Stopping criteria. Given that one cannot prove (at present) that GaussSieve terminates, stopping conditions for GaussSieve must be chosen in a heuristic way, chosen such that any further reduction in the norm of the shortest vector found is unlikely to occur. In [18], it is suggested to terminate the algorithm after a certain number of successively-sampled vectors are all reduced to zero using the extant list, with 500 such consecutive zero reductions being mentioned as a possible choice in practice. In Voulgaris' implementation, a stopping condition is employed which depends on the maximal list size encountered. In our experiments we follow the suggestions of [18] in this regard.

Complexity. As with all sieving algorithms, the complexity of GaussSieve is largely determined by arguments related to sphere packing and the *Kissing Number* - the maximum number of equivalent hyperspheres in n dimensions which are permitted to touch another equivalent hypersphere yet not intersect. With practical variants of GaussSieve, as dealt with here, no complexity bound is known due to the possibility of perpetual reductions of vectors to zero without a shortest vector being found. For further details, we direct the reader to [18].

4 Approximate Gauss Reduction

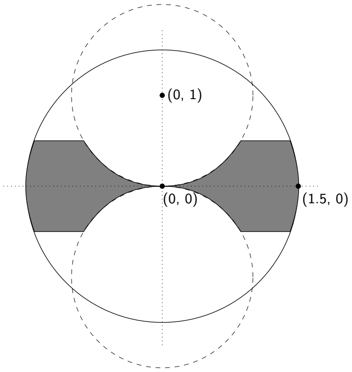

Fig. 1. Example Gauss-Reduced Region (shaded), Dimension 2.

The motivation for our first contribution stems from the observation that, at least in moderate dimension, the overwhelming majority of vector pairs we consider are already Gauss-reduced, yet we expend the vast majority of effort in the algorithm in verifying that they are indeed Gauss-reduced. Thus, by "detecting" relatively cheaply whether such a pair is *almost-certainly* Gauss-reduced, we can obtain substantial (polynomial) speedups at the cost of possibly erring (almost inconsequentially) with respect to a few pairs.

We now make an idealizing assumption, namely the random ball assumption (as appears in [25]) that we can gain insights into the behavior of lattice algorithms by assuming that lattice vectors are sampled uniformly at random from the surface of an Euclidean ball of a given radius. As in [25], we term this the "random ball model". For intuition, Figure 1 shows the region (shaded) of vectors in the ball $\mathcal{B}_2(\mathbf{0}, 1.5)$ which are Gauss-reduced with respect to the vector $(0, 1)$.

Lemma 1. *Given a vector $\mathbf{v} \in \mathbb{R}^n$ of (Euclidean) norm r sampled from at random from $\mathcal{S}_{n-1}(\mathbf{0}, r)$ and a second vector $\mathbf{w}$ sampled independently at random from $\mathcal{S}_{n-1}(\mathbf{0}, r')$ (where r' is a second radius), the probability that $\mathbf{w}$ is Gauss-reduced with respect to $\mathbf{v}$ is*

$$1 - I_{1-(r/2r')^2}\left(\frac{n-1}{2}, \frac{1}{2}\right)$$

where $h_h := r' - r/2$ and $I_x(a, b)$ denotes the regularized incomplete beta function.

Proof. We assume, without loss of generality, that $r' \geq r$, otherwise, we swap $\mathbf{v}$ and $\mathbf{w}$. The surface area of the n-sphere of radius r' (denoted $\mathcal{S}_{n-1}(\mathbf{0}, r')$) is

$$\mathcal{S}_{n-1}(\mathbf{0}, r') = \frac{n\pi^{n/2} r'^{n-1}}{\Gamma(1 + \frac{n}{2})} .$$

Then, the points from this sphere which are Gauss-reduced with respect to $\mathbf{v}$ are determined by the relative complement of $\mathcal{S}_{n-1}(\mathbf{0}, r')$ with the hyper-cylinder of radius $\sqrt{r'^2 - r^2/4}$, of which both the origin and $\mathbf{v}$ lie on the center-line. We can calculate the surface area of this relative complement by subtracting the surface area of a certain hyperspherical cap from the surface area of a hemisphere of the hypersphere of radius r'. Specifically, let us consider only one hemisphere of $\mathcal{S}_{n-1}(\mathbf{0}, r')$. Considering the hyperspherical cap of height $h_h := r' - r/2$, this cap has surface area

$$\frac{1}{2}\mathcal{S}_{n-1}(\mathbf{0}, r') I_{1-(r/2r')^2}\left(\frac{n-1}{2}, \frac{1}{2}\right) ,$$

where $I_x(a, b)$ denotes the regularized incomplete beta function:

$$I_x(a, b) := \sum_{i=a}^{\infty} \binom{a+b-1}{i} x^i (1-x)^{a+b-1-i} .$$

Thus, the relative complement has surface area

$$\frac{1}{2}\mathcal{S}_{n-1}(\mathbf{0}, r')\left(1 - I_{1-(r/2r')^2}\left(\frac{n-1}{2}, \frac{1}{2}\right)\right)$$

and hence, the probability of obtaining a Gauss-reduced vector is

$$1 - I_{1-(r/2r')^2}\left(\frac{n-1}{2}, \frac{1}{2}\right)) \ .$$

□

For instance, Figure 2 gives the probability of two vectors being *a priori* Gauss-reduced with increasing dimension in the case of $r = 1000$ and $r' = 1100$. By *a priori* Gauss-reduced, we mean that two vectors, sampled at random from zero-centered spheres of respective radii, are Gauss-reduced with respect to each other. These illustrative values are chosen to be representative of the similar-norm pairs of vectors which comprise the vast majority of attempted reductions in GaussSieve.

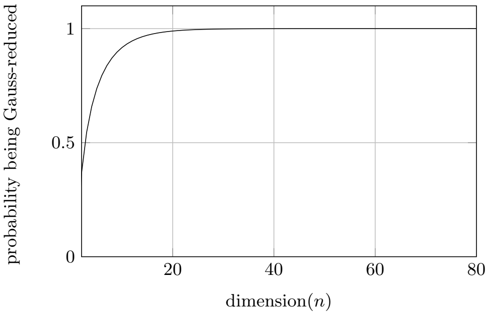

Fig. 2. Example probabilities of *a priori* Gauss-reduction, $r = 1000$, $r' = 1100$.

If we are given two such vectors, we can easily determine whether they are Gauss-reduced by considering the angle θ between them. It follows simply from elementary Euclidean geometry that if the following condition is satisfied, they are Gauss-reduced:

$$|\frac{\pi}{2} - \theta| \leq \arcsin(r/2r')$$

Thus, if we can "cheaply" determine an approximate angle, we can tell with good confidence whether they are indeed Gauss-reduced or not. We note that, while we do not believe one can prove similar arguments to the above in the context of lattices, the behavior appears indistinguishable for random lattices in practice. Indeed, we also experimented with vector pairs sampled from random lattice bases using Klein's algorithm and obtained identical behavior to that illustrated in Figures 2 and 3. For determining such approximate angles, we investigated two approaches: a) computing the angle between restrictions of vectors to subspaces and b) exploiting correlations between the XOR + population count of the sign bits of a pair of vectors and the angle between them. We only report the latter approach, which appears to offer superior results in practice.

Using XOR and Population Count as a First Approximation to the Angle. Given a vector $\mathbf{a} \in \mathbb{Z}^n$ we define $\tilde{\mathbf{a}} \in \mathbb{Z}_2^n$ such that $\tilde{\mathbf{a}}_i = \operatorname{sgn}(\mathbf{a}_i)$. Here, we define

$$\operatorname{sgn}(a) : \mathbb{R} \to \{0, 1\} \quad \text{by } \operatorname{sgn}(a) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{otherwise} \end{cases}$$

and define the normalized XOR followed by population count of $\mathbf{a}$ and $\mathbf{b}$ to be

$$\operatorname{sip}(\mathbf{a}, \mathbf{b})\colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+ \quad \text{by } \operatorname{sip}(\mathbf{a}, \mathbf{b}) = \mathrm{w}(\tilde{\mathbf{a}} \oplus \tilde{\mathbf{b}})/n$$

Based on Assumption 1, we can use the XOR + population count of $\mathbf{a}$ and $\mathbf{b}$ as a first approximation to the angle between $\mathbf{a}$ and $\mathbf{b}$ when their norms are relatively similar. The attraction of using $\operatorname{sip}(\mathbf{a}, \mathbf{b})$ as a first approximation to $\mathbf{a}\angle\mathbf{b}$ is the need to only compute an XOR of two binary vectors followed by a population count, operations which can be implemented efficiently. For intuition, consider the first components a_1, b_1 of vectors $\mathbf{a}$ and $\mathbf{b}$, respectively. If $\operatorname{sgn}(a_1) \oplus \operatorname{sgn}(b_1) = 1$ then the signs of these components are different and are the same otherwise. Clearly, in higher dimensions, when sampling uniformly at random from a zero-centered sphere, the expected number of such individual XORs would be $n/2$, hence $\mathrm{E}[\operatorname{sip}(\mathbf{a}, \mathbf{b})] = 1/2$. If $\operatorname{sip}(\mathbf{a}, \mathbf{b}) = 1$, then all components of both vectors lie in the same intersection of the sphere with a given orthant and thus we might expect that the angle between these two vectors has a good chance of being relatively small. The analogous case of $\operatorname{sip}(\mathbf{a}, \mathbf{b}) = 0$ corresponds to taking the negative of one of the vectors. Conversely, since the expected value of $\operatorname{sip}(\mathbf{a}, \mathbf{b})$ is $1/2$, we expect this to coincide with the heuristic that, in higher dimensions, most vectors sampled uniformly at random from a zero-centered sphere are almost orthogonal. Again, we stress that these arguments are given purely for intuition and appear to work well in practice, as posited in Assumption 1:

Assumption 1 *[Informal] Let $n \gg 2$. Then, given a random (full-rank) lattice Λ of dimension n and two vectors $\mathbf{a}, \mathbf{b} \in \Lambda$ of "similar" norms sampled uniformly at random from the set of all such lattice vectors, the distribution of the normalized sign XOR + population count of these vectors* $\operatorname{sip}(\mathbf{a}, \mathbf{b})$ *and the angle between them can be approximated by a bivariate Gaussian distribution.*

Note 1. We note that, in our experiments, we took "similar" norm to mean that $\max\{\|\mathbf{a}\|/\|\mathbf{b}\|, \|\mathbf{b}\|/\|\mathbf{a}\|\} \leq 1.2$, with a failure to satisfy this condition leading to full inner product calculation.

Application of Mardia's test [15] for multivariate normalcy yields confirmative results. As an example, the covariance matrix below provides a good approximation of this distribution, in dimension 96 as shown by our experiments.

$$\begin{bmatrix} 0.01200 & -0.00307 \\ -0.00307 & 0.00290 \end{bmatrix}$$

For example, Figure 3 shows the result of 100,000 pairs of vectors sampled according to a discrete Gaussian from a 96-dimensional random lattice, with the region lying between the horizontal lines containing the cases in which we assume that the pair of vectors is Gauss-reduced and hence do not expend effort in computing the full inner-product to confirm this. More specifically, we choose an integer parameter k and, when we wish to compute the angle between vectors $\mathbf{a}$ and $\mathbf{b}$, we firstly compute $c = \text{sip}(\mathbf{a}, \mathbf{b})$. If $(\lfloor n/2 \rfloor - k)/n \leq c \leq (\lceil n/2 \rceil + k)/n$ we assume that $\mathbf{a}$ and $\mathbf{b}$ are already Gauss-reduced. Otherwise, we compute $\langle \mathbf{a}, \mathbf{b} \rangle$.

Choosing k, i.e. determining the distance of the horizontal lines from $n/2$ to n was done heuristically, with values of 6 or 7 appearing to work best for the lattice dimensions with which we experimented. If k is too small, the heuristic loses value, while if it is too large we will commit too many false negatives (missed reductions) which will lead to a decreased speedup. In the case of Figure 3, $k = 6$. The occurrence of a few false negatives arising from this approach appears to have little consequence for the algorithm - this assumption appears to be borne out by the experiments reported in Section 7. We also note that false positives cannot occur.

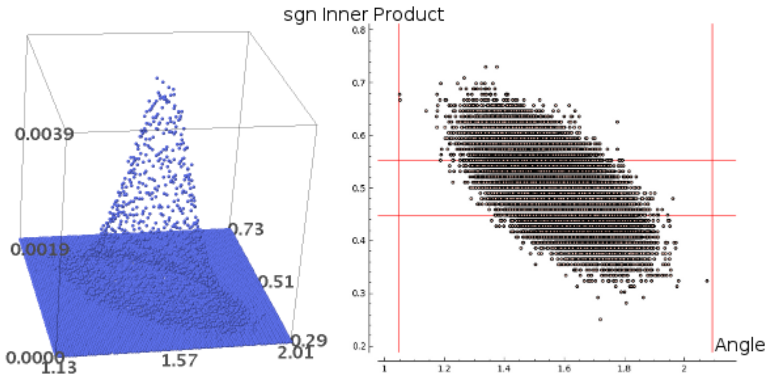

Fig. 3. Example distribution of $sip(\mathbf{a}, \mathbf{b})$ and angle between random unit vectors in dimension 96.

5 Using Multiple Randomized Bases

When examining the performance of enumeration-type algorithms in solving SVP instances, the level of preprocessing carried out on the basis is of prime importance, with the norms of the Gram-Schmidt vectors of the reduced basis being the main determinant of the running time. With sieving algorithms, however, this does not appear to hold - the level of preprocessing carried out on the basis has a far smaller impact on the running time of the sieving algorithms than might be expected at first.

We posit that a much more natural consideration is the number of randomized lattice bases which are reduced and used to "seed" the list. That is, instead of adding the input basis to the list before starting the sieving procedure, we randomize and reduce the given basis several times, appending all so-obtained lattice vectors to the list L by running GaussReduce($\mathbf{b}_i, \mathsf{L}, \mathsf{S}$) for all obtained vectors $\mathbf{b}_i$ (cf. Algorithm 1).

The idea of rerandomizing and reducing a given lattice basis for algorithmic improvements is not new. Indeed, Gama et al. [7] show, with respect to enumeration-based SVP algorithms, a theoretical exponential speedup if the input basis is rerandomized, reduced and the enumeration search tree for each reduced basis is pruned extremely. Experiments confirm this huge speedup in practice [13]. While in enumeration rerandomizing and reducing provides almost independent instances of (pruned) enumeration, in this modification to GaussSieve we instead concurrently exploit all the information gathered through all generated bases in a single instance of GaussSieve rather than running multiple instances of GaussSieve.

However, a natural concern that arises in this setting is that of the number of unique lattice vectors we can hope to obtain by way of multiple randomization and reduction - we wish for this number to be as large as possible to maximize the size of our starting list. Our experiments indicate that, given a large enough lattice dimension, the number of duplicate vectors obtained by this approach is negligible even when performing a few thousand such randomizations and reductions. Figure 4 illustrates the number of distinct vectors obtained through this approach in dimensions 40 and 70, highlighting that, beyond toy dimensions, obtaining distinct vectors through this approach is not problematic. We also observe that such a "seeding" of the list is only slightly more costly in practice as this approach makes the first stage of the algorithm embarrassingly parallel, i.e. each thread can carry out an independent basis randomization and reduction, with a relatively fast merging of the resulting collection of vectors into a pairwise Gauss-reduced list.

After seeding the list using the vectors from the reduced bases, we additionally store these bases and, rather than sampling all vectors from a single basis, sample from our multiple bases in turn. We note that our optimizations have some similarities with the random sampling algorithm of Schnorr [23]. Here, short lattice vectors are sampled to update a given basis, thereby performing multiple lattice reductions. However, we add new vectors into the list while Schnorr's algorithm uses a fixed number of vectors throughout the execution.

In practice, this modification appears to give linear speedups based on our experimental timing results given in Section 7.

Given that parallel adaptations of GaussSieve are highly practical, especially for ideal lattices, we expect the approach of randomizing and reducing the basis to seed the list to be very effective in practice. For instance, the implementation of Ishiguro et al. employed more than 2,688 threads to solve the Ideal-SVP 128-dimensional challenge, with the number of thread-hours totaling 479,904. However, only a single basis was used, having been reduced with BKZ with a

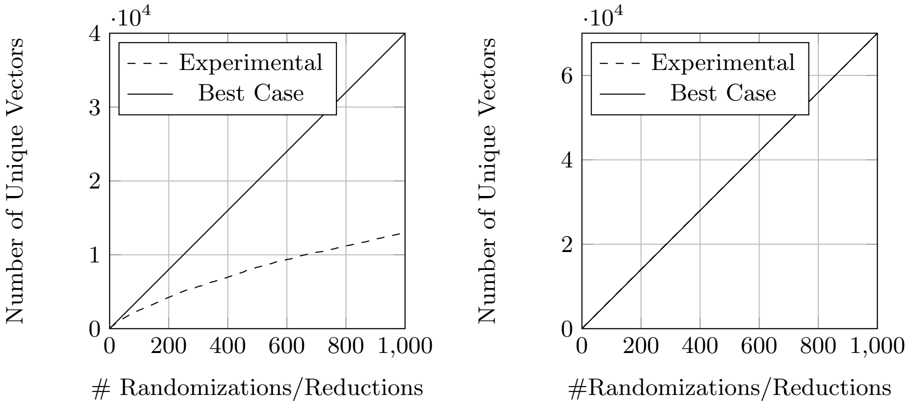

Fig. 4. Number of distinct vectors obtained under repeated (up to 1000) randomizations and BKZ reductions with blocksize 20 for random lattice in dimension 40 (left) and dimension 70 (right). In dimension 70 the two lines coincide almost exactly.

blocksize of 30. If each thread additionally performed, say three randomizations and reductions, over one million unique lattice vectors would be easily obtained. In comparison, in dimension 128, we would expect our final list to contain roughly 4.2 million vectors in the ideal lattice case.

6 Reducing the Gaussian Parameter

Recall that Klein's algorithm samples from a discrete Gaussian over the given lattice by taking (integer) linear combinations of the basis vectors, with each coefficient being sampled from a discrete Gaussian of parameter s over $\mathbb{Z}$. The parameter s is proportional to the norm of the Gram-Schmidt vector corresponding to that dimension. Unfortunately, in the implementation of Voulgaris [1], Klein's algorithm is implemented incorrectly, with the result that one either samples integers which are non-Gaussian, or one does sample from a Gaussian but very slowly.

In the original implementation of Voulgaris, an arbitrary Gaussian sampling parameter is chosen, while Ishiguro et al. choose an arbitrary though smaller parameter. We choose the Gaussian parameter dynamically in our experiments, i.e., by starting with an unfeasibly small (for example 500) parameter (which is guaranteed to return only the zero vector) and then incrementing this by one each time Klein's algorithm returns a zero vector. The intuition for this strategy is that, if the Gaussian parameter is too large, Klein's algorithm will generate unnecessarily long vectors, while if the Gaussian parameter is too small, the only vectors delivered by Klein's algorithm will be the zero vector and (occasionally) single vectors from the basis. Hence we need to choose a Gaussian parameter which is large enough that the number of lattice vectors obtainable is large

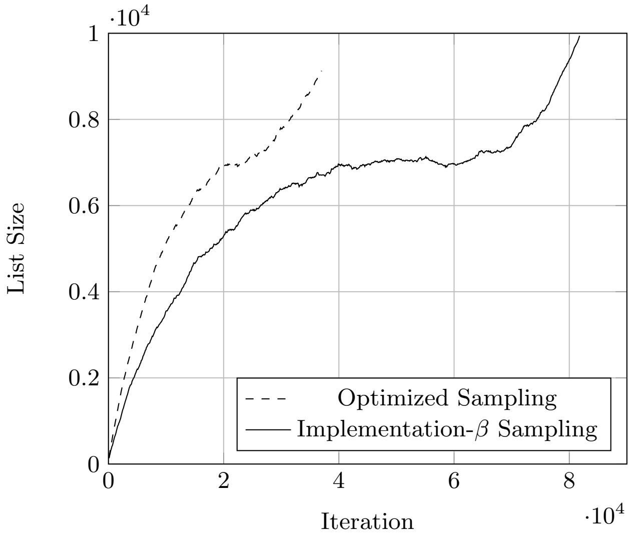

Fig. 5. Progressive list sizes in a 56-dimensional random lattice.

enough to generate a list which satisfies the termination condition, but which is small enough that the vectors generated are not too long as to impose an additional unnecessary number of iterations on the algorithm. While it is probably possible to prove an optimal value for the Gaussian parameter (i.e. to provide a lower-bound on the Gaussian parameter which leads to enough entropy in Klein's algorithm to deliver a terminating list), we do not deal with this here as, in practice, our approach of dynamically increasing the parameter upon seeing zero vectors appears to work very well.

Upon choosing the parameter in this way, a substantial change in behavior occurs, as illustrated in Figure 5, with far fewer iterations (less than half as many in this case) being necessary to satisfy the termination condition. In contrast to the suggestions in Ishiguro et al. (in which it was suggested that a fixed Gaussian parameter should be used throughout the execution of the algorithm but that the optimal Gaussian parameter increases with lattice dimension), our experiments indicate that decreasing the Gaussian sampling parameter with increasing dimension delivers superior results[1].

7 Implementation and Experimental Results

To test the modifications outlined, we adapted the single-threaded implementation of Voulgaris [1], comparing minimally-modified versions to the reference

[1] We note that the speedups gained from dynamic choice of the Gaussian parameter are independent of the bug in the reference implementation, said bug leading to only a minor slowdown in most cases. See Table 1 further for details.

implementation. While several obvious optimizations are possible, we did not implement these, for consistency. We stress, however, that the timings given here are purely for comparative purposes and in addition to our algorithmic optimizations further optimizations at the implementation level can significantly enhance the performance of the algorithm, for instance using 16-bit integers for vector entries rather than the 64-bit integers used in the reference implementation.

All experiments were carried out using a single core of an AMD FX-8350 4.0GHz CPU, 32GB RAM, with all software (C++) compiled using the Gnu Compiler Collection, version 4.7.2-5. Throughout, we only experiment with the Goldstein-Mayer quasi-random lattices as provided by the TU Darmstadt SVP challenge [2].

7.1 Our Timings

In order to better assess the impact of our modifications to GaussSieve, we compare our implementations both to the original implementation of Voulgaris and a "corrected" version where we embed a correct implementation of a discrete Gaussian sampler[2]. We denote the original implementation by "Reference Implementation" and the original implementation + corrected Gaussian sampler by "Reference Implementation-β".

Table 1 shows timings for the original (unoptimized) implementation of Voulgaris [1], of Reference Implementation-β, and of our proposed optimizations explicitly. We also provide timings for an implementation which incorporates all the discussed optimizations for which the pseudocode can be found in Appendix A. For the multiple-bases optimization we display the timing with best efficiency, i.e., with the optimal (in terms of our limited experiments) number of bases. All timings exclude the cost of lattice reduction but we include the additional necessary lattice reduction via BKZ when considering multiple bases.

We observe that our optimized Gaussian sampler gives a speedup of up to 3.0x. However, with increasing dimension the speedup decreases slightly. Our integration of approximate inner-product computations increases performance by a factor of up to 2.7x, as compared to the original implementation of GaussSieve.

Similar speedups are obtained by considering multiple randomized bases; however, the speedup increases for larger dimensions. Indeed, if we ignore dimension 70, for which we did not consider an optimal number of bases due to time constraints, the speedup is approximated closely by the function $0.1838n - 9.471$. Figure 6 illustrates the speedups for several dimensions when increasing the number of bases considered.

When employing multiple randomized bases it is almost always the case that with increasing dimension employing more bases is preferable. Table 2 shows the

[2] In the implementation of Voulgaris, no lookup table is employed for Gaussian carrying out rejection sampling over a subset of the integers. Hence, the sampled integers are much closer to uniform than to the intended truncated Gaussian. In our corrected comparative implementation we employ the same Gaussian parameter from the Voulgaris implementation but ensure that the sampled vectors adhere to the prescribed Gaussian.

Dimension	60	62	64	66	68	70
Reference Implementation [1]	464	1087	2526	5302	12052	23933
Reference Implementation-β	455	1059	2497	5370	12047	24055
XOR + Pop. Count (Sec. 4)	203	459	1042	2004	4965	11161
Mult. Rand. Bases (Sec. 5)	210	555	1103	2023	3949	7917
Opt. Gaussian Sampling (Sec. 6)	158	376	1023	2222	5389	10207
Combined (s)	79	146	397	868	2082	4500
Shortest Norm $\approx$	1943	2092	2103	2099	2141	2143

Table 1. Execution time (in seconds) of Voulgaris' implementation [1] and our optimized variants.

runtime of our implementation when employing various numbers of randomized bases. It also depicts the amount of time necessary to reduce all the generated bases.

Dim.	Number of Additional Bases						
	0	10	20	40	80	160	320
60	(453, 0)	(274, 2)	(238, 4)	(210, 8)	(195, 15)	(185, 29)	(164, 59)
62	(1075, 0)	(810, 1)	(686, 3)	(612, 12)	(570, 12)	(533, 22)	(530, 43)
64	(2507, 0)	(1389, 17)	(1209, 36)	(1025, 75)	(877, 153)	(723, 322)	(461, 748)
66	(5302, 0)	(3193, 19)	(2716, 41)	(2328, 83)	(1961, 171)	(1659, 364)	(1233, 835)
68	(12052, 0)	(6842, 23)	(5852, 48)	(5071, 99)	(4360, 200)	(3652, 415)	(3015, 934)
70	(23933, 0)	(14933, 24)	(12641, 53)	(10933, 111)	(9561, 225)	(8139, 464)	(6871, 1046)

Table 2. Time for (sieving, initialization) in seconds.

Acknowledgments

The authors would like to thank the anonymous reviewers of Latincrypt 2014 for their helpful comments and suggestions which substantially improved this paper. Özgür Dagdelen is supported by the German Federal Ministry of Education and Research (BMBF) within EC-SPRIDE.

References

1. Panagiotis Voulgaris - GaussSieve Implementation. `http://cseweb.ucsd.edu/~pvoulgar/impl.html`.
2. TU Darmstadt Lattice Challenge. `http://www.latticechallenge.org`.
3. Miklós Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *STOC '98*, pages 10–19, NY, USA, 1998. ACM.

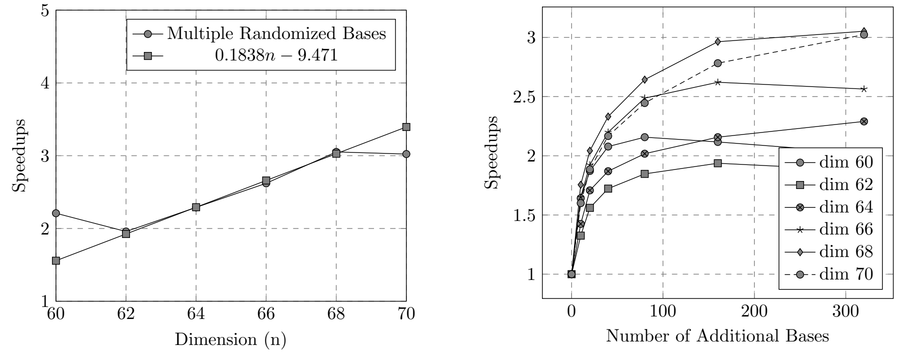

Fig. 6. Speedups with respect to the dimension (left) and the number of additional bases used to seed the list before sieving (right).

4. Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *STOC*, pages 601–610. ACM, 2001.
5. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better Lattice Security Estimates. In *ASIACRYPT*, pages 1–20, 2011.
6. Nicolas Gama and Phong Q. Nguyen. Predicting Lattice Reduction. In *EUROCRYPT*, pages 31–51, 2008.
7. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice Enumeration Using Extreme Pruning. In *EUROCRYPT*, pages 257–278, 2010.
8. Nicolas Gama and Michael Schneider. SVP Challenge, 2010. available at `http://www.latticechallenge.org/svp-challenge`.
9. Daniel Goldstein and Andrew Mayer. On the Equidistribution of Hecke Points. In *Forum Mathematicum*, volume 15, pages 165–190, 2003.
10. Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel Gauss Sieve Algorithm: Solving the SVP Challenge over a 128-Dimensional Ideal Lattice. In *Public Key Cryptography*, pages 411–428, 2014.
11. Ravi Kannan. Improved Algorithms for Integer Programming and Related Lattice Problems. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *STOC*, pages 193–206. ACM, 1983.
12. Philip N. Klein. Finding the Closest Lattice Vector when it's Unusually Close. In *SODA*, pages 937–941, 2000.
13. Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme Enumeration on GPU and in Clouds – How Many Dollars You Need to Break SVP Challenges. In *CHES*, pages 176–191, 2011.
14. Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring Polynomials with Rational Coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.
15. K. V. Mardia, editor. *Tests of Univariate and Multivariate Normality.* Handbook of Statistics. North-Holland, 1980.
16. A. Mariano, Shahar Timnat, and Christian Bischof. Lock-free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation. In *SBAC-PAD*, 2014.

17. D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective.* Milken Institute Series on Financial Innovation and Economic Growth. Springer US, 2002.
18. Daniele Micciancio and Panagiotis Voulgaris. Faster Exponential Time Algorithms for the Shortest Vector Problem. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1468–1480. Society for Industrial and Applied Mathematics, 2010.
19. Benjamin Milde and Michael Schneider. A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices. In Victor Malyshkin, editor, *PaCT*, volume 6873 of *Lecture Notes in Computer Science*, pages 452–458. Springer, 2011.
20. Phong Q. Nguyen and Thomas Vidick. Sieve Algorithms for the Shortest Vector Problem are Practical. *J. Mathematical Cryptology*, 2(2):181–207, 2008.
21. Michael Schneider. Sieving for Shortest Vectors in Ideal Lattices. *IACR Cryptology ePrint Archive*, 2011:458, 2011.
22. Claus-Peter Schnorr. A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987.
23. Claus-Peter Schnorr. Lattice Reduction by Random Sampling and Birthday Methods. In *STACS*, pages 145–156, 2003.
24. Carl Ludwig Siegel. A mean value theorem in geometry of numbers. *Annals of Mathematics*, 46(2), 1945.
25. Brigitte Vallée and Antonio Vera. Probabilistic Analyses of Lattice Reduction Algorithms. In Phong Q. Nguyen and Brigitte Vallée, editors, *The LLL Algorithm*, Information Security and Cryptography, pages 71–143. Springer, 2010.
26. Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In *ASIACCS*, pages 1–9, 2011.
27. Feng Zhang, Yanbin Pan, and Gengran Hu. A Three-Level Sieve Algorithm for the Shortest Vector Problem. In *Selected Areas in Cryptography*, pages 29–47, 2013.

A Pseudocode of our Optimized GaussSieve

The below pseudocode displays our proposed modifications to GaussSieve. In lines (3)-(9) we incorporate our multiple-randomized-bases optimization, and in the function `GaussReduce`($\mathbf{p}$,L, S, k) we embed the cheap test *SIP* implementing our XOR + population count computation for the approximation of the angle between two vectors. The optimized Gaussian sampler modifies the function `SampleKlein`.

In the pseudocode, the parameter $k \in \mathbb{Z}^+$ defines the bounds on the XOR + population count, within which we assume that a pair of vectors is Gauss-reduced, i.e. if $n/2 - k \leq \langle \tilde{\mathbf{a}}, \tilde{\mathbf{b}} \rangle \leq n/2 + k$, we assume the pair $\mathbf{a}, \mathbf{b}$ are Gauss-reduced.

Algorithm 2: Optimized GaussSieve

```
Input  : Basis $\mathbf{B}$, $k \in \mathbb{Z}^+$, $r \in \mathbb{Z}^+$,
         $a' \in \mathbb{R}^+$, $\delta \in \mathbb{R}^+$
Output: $\mathbf{v} : \mathbf{v} \in \Lambda(\mathbf{B}) \wedge \|\mathbf{v}\| = \lambda_1(\mathbf{B})$
$\mathsf{L} \leftarrow \{\}, \mathsf{S} \leftarrow \{\}, \mathsf{col} \leftarrow 0, a \leftarrow a'$
repeat
    $\mathbf{B}' \leftarrow$ RandomizeBasis($\mathbf{B}$)
    for $\mathbf{v} \in \mathbf{B}'$ do
        v' ← GaussReduce($\mathbf{v}$,$\mathsf{L}$, $\mathsf{S}$, $k$)
        if $\|\mathbf{v}'\| \neq 0$ then
            $\mathsf{L} \leftarrow \mathsf{L} \cup \{\mathbf{v}\}$
until $r$ times
while $\mathsf{col} < c$ do
    if $\mathsf{S}$ is not empty then
        $\mathbf{v} \leftarrow \mathsf{S}$.pop()
    else
        $\mathbf{v} \leftarrow$ SampleKlein($\mathbf{B}$', a)
        while $\mathbf{v} = \mathbf{0}$ do
            $a \leftarrow a + \delta$
            $\mathbf{v} \leftarrow$ SampleKlein($\mathbf{B}$', a)
    j ← GaussReduce($\mathbf{v}$,$\mathsf{L}$, $\mathsf{S}$, $k$)
    if $j$ = true then
        $\mathsf{col} \leftarrow \mathsf{col} + 1$
return $\mathbf{v} \in \mathsf{L}$ s.t. $\|v\| = \min_{\mathbf{x} \in \mathsf{L}} \|\mathbf{x}\|$

function GaussReduce($\mathbf{p}$,$\mathsf{L}$, $\mathsf{S}$, $k$)
was_reduced ← true
while was_reduced = $true$ do
    was_reduced ← false
    for all $\mathbf{v}_i \in \mathsf{L}$ do
        if SIP($\mathbf{v}_i, \mathbf{p}, k$) = 1 then
            if $\exists t \in \mathbb{Z}$: $\|\mathbf{p} + t\mathbf{v}_i\| < \|\mathbf{p}\|$ then
                $\mathbf{p} \leftarrow \mathbf{p} + t\mathbf{v}_i$
                was_reduced ← true
if $\|\mathbf{p}\| = 0$ then
    return true
for all $\mathbf{v}_i \in \mathsf{L}$ do
    if SIP($\mathbf{v}_i, \mathbf{p}, k$) = 1 then
        if $\exists u \in \mathbb{Z}$: $\|\mathbf{v}_i + u\mathbf{p}\| < \|\mathbf{v}_i\|$ then
            $\mathsf{L} \leftarrow \mathsf{L} \backslash \{\mathbf{v}_i\}$
            $\mathbf{v}_i \leftarrow \mathbf{v}_i + u\mathbf{p}$
            $\mathsf{S}$.push($\mathbf{v}_i$)
$\mathsf{L} \leftarrow \mathsf{L} \cup \{\mathbf{p}\}$
return false
end function
```