A new Parallelization for p3Enum and Parallelized Generation of Optimized Pruning Functions

Michael Burger
Scientific Computing
TU Darmstadt
Darmstadt, Germany
michael.burger@sc.tu-darmstadt.de

Christian Bischof
Scientific Computing
TU Darmstadt
Darmstadt, Germany
christian.bischof@sc.tu-darmstadt.de

Juliane Krämer
Cryptography and Computer Algebra
TU Darmstadt
Darmstadt, Germany
j.kraemer@cdc.informatik.tu-darmstadt.de

***Abstract*—Since quantum computers will be able to break all public-key encryption schemes employed today efficiently, quantum-safe cryptographic alternatives are required. One group of candidates are lattice-based schemes since they are efficient and versatile. To make them practical, their security level must be assessed on classical HPC systems in order to determine efficient but secure parameterization.**

In this paper, we propose a novel parallelization strategy for the open source framework p3Enum which is designed to solve the important lattice problem of finding the shortest non-zero vector in a lattice (SVP). We also present the p3Enum extreme pruning function generator (p3Enum-epfg) which generates optimized extreme pruning functions for p3Enum's pruned lattice enumeration by employing a parallelized simulated annealing approach. We demonstrate the quality of the pruning functions delivered. Combining the new parallelization with optimized pruning functions speeds up p3Enum by a factor up to 3 compared to the previous version.

Additionally, we compare the required runtime to solve the SVPs with state-of-the art tools and, for the first time, also visualize the statistical effects in the runtime of the algorithms under consideration. This allows a considerably better understanding of the behavior of the implementations than previous average-value considerations and demonstrates the relative stability of p3Enum's parallel runtimes which improve reproducibility and predictability. All these advancements make it the fastest SVP solver for lattice dimensions 66 to 92 and a suitable building block as SVP-oracle in lattice basis reduction.

***Index Terms*—Lattice-based cryptography, Extreme pruning, OpenMP, Parallel lattice enumeration, Parallel simulated annealing, Heuristic optimization**

I. Introduction

To secure applications like mobile phone calls or messaging, reliable and secure cryptography is required. With the internet of things and future developments like autonomous cars, the need for strong cryptography grows further. While we have good cryptographic algorithms that have withstood the classical cryptanalysis since many decades, we know that all public-key cryptography in use today will be broken once large-scale quantum computers exist [1]. Thus, quantum-safe alternatives are required. One promising candidate is lattice-based cryptography. One main problem for lattices is the shortest vector problem (SVP) where one searches the shortest non-zero vector in a lattice. Many other lattice problems can be transformed to SVP instances. Hence, knowledge about the actual hardness of the SVP is crucial. Two prominent approaches to solve SVPs are enumeration with extreme pruning [2]–[5] and sieving [6]–[8]. To develop secure but efficient and practical cryptographic schemes, appropriate parameterizations, e.g,. key sizes, have to be determined on classical hardware. Thus, cryptanalysis tries to perform attacks on such schemes on available HPC systems and efficient parallel implementations of attacking algorithms are required. The main challenge in this field is that the problems secure cryptographic schemes rely on are not solvable in reasonable runtime. Otherwise the scheme would be insecure. Hence, research targets on solving problems of smaller size, in the case of lattices with lower dimensions, as fast as possible and to extrapolate the knowledge gained to higher dimensions to choose secure but efficient parameterization for the instantiations of the cryptographic schemes. An example for a software that is employed in this context is the open source framework p3Enum [5] implementing shared memory parallelized enumeration with extreme pruning to solve the SVP.

In this paper, we propose two valuable extensions to p3Enum. First we develop an alternative parallelization approach for the extreme pruning algorithm and second we implement the extreme pruning function generator (p3Enum-epfg) module which creates optimized pruning functions for the novel parallelization. The pruning function generation is based on a shared memory parallelized simulated annealing algorithm and is highly configurable. In combination, both extensions speed up p3Enum by a factor up to 3 and increase the range of lattice dimensions for which p3Enum is the fastest solver to $\{66, \ldots, 94\}$. As a second contribution, we perform for the first time a detailed performance comparison of state-of-the-art SVP solvers based on enumeration with extreme pruning and sieving and analyze the statistical effects due to their inherent algorithmic randomness. This results in a considerably better insight on the practical behavior of the solvers than previous average case considerations. We show the relative stability in p3Enum's parallel runtimes enabling better reproducibility and predictability.

This work has been co-funded by the DFG as part of project P1 within the CRC 1119 CROSSING and BI 714/6-1.

II. Preliminaries and Definitions

A. Lattices and basis reduction

We denote vectors with bold lower case letters, e.g., $\mathbf{u}$, matrices with bold upper case letters, e.g., $\mathbf{B}$, and scalars with normal lower case letters, e.g., β. $\mathbf{B}^{m\times n}$ stands for an $m \times n$ matrix. If the dimensions are clear from the context, we simply write $\mathbf{B}$. Integers are denoted by $\mathbb{Z}$ and the real numbers by $\mathbb{R}$. The standard inner product is denoted by $\langle\cdot,\cdot\rangle$ and the Euclidean norm by $\|\cdot\|$.

A lattice of dimension d is a discrete additive subgroup of $\mathbb{R}^d$. Every lattice $\Lambda \subset \mathbb{R}^d$ can be represented by a basis, i.e., a set of $\mathbb{R}$-linearly independent vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subset \mathbb{R}^d$ such that $\Lambda = \Lambda(\mathbf{B}) = \mathbb{Z}\mathbf{b}_1 + \cdots + \mathbb{Z}\mathbf{b}_n$. We identify lattice bases with matrices whose columns represent the basis vectors. In this case, d is called the dimension of the lattice and $n \leq d$ is called its rank. If $n = d$, the lattice is called a full-rank lattice. All lattices within this work are full-rank and their determinant or volume is given by $\det(\Lambda) = |\det(\mathbf{B})|$ for any basis $\mathbf{B}$ of Λ. The Gram-Schmidt (GS) basis (obtained by GS orthogonalization) of a basis $\mathbf{B}$ is denoted by $\mathbf{B}^* = \{\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*\} \subset \mathbb{R}^d$, the respective GS-lengths by $||\mathbf{b}_1^*||^2, \ldots, ||\mathbf{b}_n^*||^2$, and the GS-coefficients by $\mu_{i,j}$ with $1 \leq j < i \leq n$. The orthogonal projection to the span of $(\mathbf{b}_1, \ldots, \mathbf{b}_i)$ is denoted by π_i.

The quality of a lattice basis $\mathbf{B}$ can, e.g., be measured by the decrease of the series $||\mathbf{b}_1^*||^2, \ldots, ||\mathbf{b}_n^*||^2$, or by the value of $||\mathbf{b}_1^*||^2$. The process of improving the quality of a basis is called basis reduction. Geometrically, basis reduction means, in particular, to make the basis vectors shorter and more orthogonal. The most commonly used basis reduction algorithm is BKZ 2.0 [4]. BKZ 2.0 works on local blocks of lattices of dimension $\beta < d$ and optimizes the basis by sliding over all basis vectors in contiguous blocks. A basis processed by BKZ with block size β is called BKZ-β reduced. Solutions for the SVP or approximately good solutions, delivered by a so-called SVP-oracle, are required within each local block. Since the SVP-oracle may be called several thousand times its performance and predictability is crucial.

B. Basics of enumeration

ENUM, which p3Enum is based on, was presented by Schnorr and Euchner [9] in 1994. It fully enumerates all integer coefficient vectors $\mathbf{u} = (u_1, \ldots, u_n)$ which fulfill $||\mathbf{u} * \mathbf{B}|| < \bar{A}$ where $\bar{A}$ is an upper bound for the length of the shortest vector. Its runtime mainly depends on two factors. The choice of the bound $\bar{A}$ (the tighter, the faster) and the quality of the reduced basis where a higher quality of $\mathbf{B}$ reduces the search space.

Algorithm 1 gives a high level overview of *ENUM*. The vector $\mathbf{l}$ keeps the lengths resulting from the partial coefficient vectors while $\mathbf{c}$ holds the center-values for each level which result from the choices for the previous entries in $\mathbf{u}$. Center means that other values for u_i at level i are searched symmetrically around u_i. One main advantage of *ENUM* is that its memory requirements only grow polynomially in d.

Algorithm 1: The ENUM algorithm.

```
Input  : GS-lengths: $||\mathbf{b}_1^*||^2, \ldots, ||\mathbf{b}_n^*||^2$, GS-coefficients: $\mu_{i,j}$
Output : Coefficients of $\mathbf{u}_{min}$
1  $\bar{A} \leftarrow ||\mathbf{b}_1^*||^2$, $\mathbf{u}_{min} = \mathbf{u} = (1, 0, \ldots, 0)$, $\mathbf{l} = \mathbf{c} = \mathbf{0}$, t = 1
2  while $t \leq n$ do
3      $l_t \leftarrow l_{t+1} + (u_t + c_t)^2 ||\mathbf{b}_n^*||^2$
4      if $l_t < \bar{A}$ then
5          if $t > 1$ then
6              $t \leftarrow t - 1$
7              $c_t \leftarrow \sum_{i=t+1}^{n} u_i \mu_{i,t}$, $u_t \leftarrow \lfloor c_t \rceil$
8          else
9              $\bar{A} \leftarrow l_t$, $\mathbf{u}_{min} = \mathbf{u}$
10     else
11         $t \leftarrow t + 1$, choose next value for $u_t$
```

In essence, Algorithm 1 describes a depth-first traversal of a weighted tree, where each level corresponds to one entry of $\mathbf{u}$. The tree is flipped compared to the order of the vector, i.e., its deepest level corresponds to the first entry u_1 of $\mathbf{u}$. In Line 1, we set our initial solution to $(1, 0, \ldots, 0)$ and determine that we start our search at the leftmost entry of $\mathbf{u}$. If the resulting partial length in $\mathbf{l}$ at some level exceeds the bound $\bar{A}$, the level t is increased in the direction of the root and a new value for u_{t+1} is set following the so-called zigzag pattern. This means if we have a center value c_{t+1} for level $t + 1$, we first try the search with c_{t+1}, then with $c_{t+1} \pm 1$, then $c_{t+1} \pm 2$ and so on. To keep track of this pattern two additional vectors of length n are required. $\lfloor c_t \rceil$ in Line 7 rounds c_t to the nearest integer.

One important improvement to enumeration is the so-called pruning. The concept of pruning is that instead of traversing the whole weighted tree, one heuristically cuts off parts of the tree where the shortest vector is unlikely to be found. Technically, pruning is realized by replacing the constant $\bar{A}$ of Algorithm 1 by a vector $\mathbf{A} = (A_1, \ldots, A_n)$ with $A_1 \leq A_2 \leq \cdots \leq A_n \leq \bar{A}$. Each A_i represents the maximum value of l_i for each level i.

The idea of pruning was already mentioned in [9]. The proposed strategy is defined as follows: $A_{n-i} = \bar{A} \cdot \frac{i}{n}$ and is called linear pruning where the chance of keeping the shortest vector in the pruned tree is relatively high.

Extreme pruning was proposed by Gama et al. [2] where large parts of the enumeration tree are cut off and the chance of removing the shortest vector is high. Hence, many repetitions have to be performed and $\mathbf{B}$ is randomized each time. $\mathbf{A}$ is based on heuristic assumptions (see below). Gama et al. [2] demonstrated that in this way SVPs can be solved faster by several orders of magnitude. In the remainder of this paper pruned enumeration always indicates enumeration with extreme pruning for brevity and $\mathbf{A}$ denotes the vector containing the A_i entries and at the same time $\mathbf{A}$ may be referred as the *(extreme) pruning function* $\mathbf{A}$. To generate $\mathbf{A}$, [2] employs heuristics to estimate the length of the shortest vector $\bar{A}$, the values of the GS-lengths $||\mathbf{b}_i^*||^2$ and the runtime of BKZ t_{BKZ}. They also estimate the number of nodes to be visited in the enumeration tree and measure the average time to process one node on their system. The product of both gives the runtime for one enumeration t_{ENUM}. Together with an estimate for the probability p of success for a single enumeration, the overall runtime is calculated by $\frac{t_{BKZ}+t_{ENUM}}{p}$. This term is heuristically optimized by changing $\mathbf{A}$ which influences all three variables t_{BKZ}, t_{ENUM} and p.

Details about sieving, the second class of algorithms whose performance is also considered in this paper can be found in [6]–[8].

III. Related Work

In this section, we present other SVP solvers based on pruned enumeration and sieving as well as generators for extreme pruning functions. Their performance is compared to our extended p3Enum-version in Section V.

The template-based fplll library [10] is written in C++ and implements important algorithms from the lattice domain like LLL, BKZ 2.0, or solving SVPs with pruned enumeration. The pruned enumerations are executed with one thread, based on values provided by a strategy file which contains the pruning function and determines the sequence of β-values for the BKZ reduction. It is limited to dimension 90. So, pruned enumeration is not possible for $d > 90$ by default. The additional program *fplll strategizer* generates pruning functions and the BKZ-β-strategies.

Kuo et al. [3] presented an implementation of extreme pruning on GPUs. Their basic parallelization approach is taken from Hermans et al. [11]. The enumeration tree is split into two parts: The CPU searches within the first few dimensions for coefficient vectors with a feasible length. Those starting vectors are transferred to the GPUs which finish the vectors by enumerating the remaining dimensions. The pruning function results from sampling the exemplary bounding function visualized in [2, Fig. 1] as a polynomial of degree eight and scaling it to the dimension required. In [3], dimensions 80 to 104 are solved on a workstation with eight NVIDIA GTX480 cards.

Aono et al. [4] developed progressive BKZ, an extended version of BKZ 2.0. For the pre-processing of the blocks, no predefined BKZ-strategy is used as, e.g., it is provided in the fplll library by the strategy file. Instead, they apply a progressive approach which starts with a small β value and iteratively increases the block size in appropriate steps. They also improve the value of the initial enumeration bound $\overline{A}$ by optimizing the estimates with geometric series assumption. The model for the overall runtime is changed to the so-called full enumeration cost (FEC) which influences all automatic parameter choices and the termination criterion. The FEC is additionally based on a benchmark. Aono et al. [4] published code implementing their methods (pBKZ-lib)[1] which is mainly meant to reproduce the published results and is not designed as a library for lattice problems.

[1]https://www2.nict.go.jp/security/pbkzcode/

Concerning sieving, [6] and [7] simultaneously proposed the idea of progressive sieving. Instead of directly solving the SVP on the whole lattice basis, the process starts on sublattices $\Lambda_{[0,i]}$. These $\Lambda_{[0,i]}$ result from the projection of the basis $\mathbf{B}_{[0,i]}$, for instance with $i = n/2$ in [7]. In [6], a speedup up to a factor of 5000 is achieved when extending their sieving implementations with this new approach.

Ducas [7] proposes a second improvement for sieving which results in a sub-exponential gain in the overall runtime. It takes advantage of the fact that the output of sieving is not a single vector shorter than the bound, but a whole list of short vectors. This allows to solve the n-dimensional SVP with several sieving calls on $(n-\delta)$-dimensional sublattices, where δ is heuristically determined. As an example, [7, Fig. 2] shows that for an 82-dimensional lattice on average only sieving calls in dimension 68 are required. The resulting sieving procedure is called SubSieve[2].

Recent work of Albrecht et. al [8] combines all principles of SubSieve with further algorithmic improvements into the General Sieve Kernel (G6K). The authors call G6K an abstract stateful machine with some basic instructions that are sufficient to encode the whole procedure. One main contribution of this work is that it solves the following problem: If a basis is processed in blocks, then how is it possible to take advantage of the information gained by one block in the following one? So, they do not have to process contiguous blocks when reducing the basis. If the algorithm processed block $[\mathbf{b}_i, \mathbf{b}_j]$ with $1 \leq i < j \leq n$, the next processed block may be $\left[\mathbf{b}_{i+k}, \mathbf{b}_{max(j+k,n)}\right]$ with $k > 1$. Albrecht et. al [8] describe a parallelized C++-implementation with a Python-based control module for G6K and evaluate its performance in detail. The code solves previously unsolved dimensions 153 and 155 in the Darmstadt SVP challenge (D-SVPC)[3], where researchers are invited to try to find short vectors within provided random lattices to compare various algorithmic approaches and is the fastest algorithm for higher dimensions so far. The code has recently been made available[4].

IV. Software Architecture of p3Enum and p3Enum-epfg

A. Parallelization strategy for the SVP solver

In contrast to the original p3Enum-version and the approaches in the literature [3], [4], our extension does not parallelize the enumeration itself, but executes several basis reduction-enumeration-cycles in parallel. On the entry of the function `solveSVP()` an OpenMP `parallel`-region is spawned. Within an OpenMP `single`-block the number of active threads for `solveSVP()` is determined and shared data structures are allocated like the vector for randomized bases, the shortest solution found per thread and a flag indicating to stop the work. Afterward, each thread allocates its private data structures like the GS-lengths $||\mathbf{b}_1^*||^2, \ldots, ||\mathbf{b}_n^*||^2$ vector or the

[2]https://github.com/lducas/SubSieve

[3]https://www.latticechallenge.org/svp-challenge/

[4]https://github.com/fplll/g6k

matrix for the GS-coefficients $\mu_{i,j}$. Then, each thread enters a `while`-loop which executes until the shortest vector is found or the maximum number of trials is reached.

Per trial each thread randomizes the input base and reduces it with two calls to the fplll library where the block size of the first call of BKZ 1.0 is determined by the parameter pre-β and the blocksize of the second call of BKZ 2.0 by β. The randomization of the bases also takes the ID of the executing thread into account to maximize the degree of randomness between the different threads. Afterward, each thread executes the enumeration on his basis. Hence, the level of parallelization is shifted to one level higher. After each cycle, the threads check if the solution was found and terminate in that case, meaning that all other threads will finish their cycle but the shortest vector is already outputted. They additionally increment the counter for the trials performed within an OpenMP `atomic`. This minimizes the communication between the threads and increases the parallel scalability.

In that way, we avoid the problem that the error between the estimated number of nodes in the search tree (c.f. Section IV-B2) and the actually visited nodes grows when executing the parallelized enumeration because more nodes are visited to create candidates which result in dead ends. Additionally, the parameterization for the enumeration process is considerably simplified since in contrast to the former version, no manual parameter setting, like for the height of the serial part of the enumeration or the shift of the pruning function (cf. [5]), is necessary anymore. This guarantees efficiency without additional effort by the user. Furthermore, the number of shared data structures is reduced which simplifies the use of additional degrees of parallelization through accelerators or several compute nodes.

B. p3Enum-epfg

In this section, we first explain our design to estimate the runtime for solving SVPs based on the original estimate from [2] $c = \frac{t_{BKZ}+t_{ENUM}}{p}$ and second describe our mapping between the extreme pruning function domain and simulated annealing as well as the parallel implementation of this algorithm.

1) Basis reduction benchmarks: In contrast to [4] and [2], the times for BKZ-reduction t_{BKZ} of the input bases are not determined by heuristics but the reductions are executed and their runtime is measured. The idea behind this procedure is that the reductions are performed several hundred or thousand times during solving SVPs of the corresponding lattice dimension. Hence, some benchmarking runs do not have a considerable effect on the overall runtime and the generation of the pruning functions is a pre-processing step which is executed once.

The benchmarking of BKZ is highly parameterizable. First, the quadruple (pre-β_{start}, pre-β_{end}, β_{start}, β_{end}) determines the search space for the BKZ-calls. All possible combinations of pre-β and β with $\beta >$ pre-β are performed successively with a step size of two for the pre-β and β values. One unique pair (pre-β, β) is called a β-configuration. The second parameter `ann_parallel_reducing_threads` determines how many threads are spawned to reduce the bases in parallel during the benchmark and later during the execution of the SVP solution process. The time required as well as the resulting values for the $||\mathbf{b}_i^*||^2$ are logged in a file. When timing values for a given β-configuration and the same value of `ann_parallel_reducing_threads` are available on the file system, they are automatically loaded and complemented with those β-configurations that have to be computed on-the-fly.

As a third parameter, `ann_bkz_instances` determines how many bases for one dimension are considered during the benchmark process and later in the annealing optimization procedure. When setting `ann_bkz_instances` to one, only the original input base is employed. If `ann_bkz_instances > 1` the parameter `ann_num_different_bases` comes into play. It determines how many different bases with different bases, e.g,. other random matrices with different seeds of the same dimension, should be considered. If `ann_bkz_instances > ann_num_different_bases` then randomized instances of the bases are created as they later appear in the pruned enumeration cycles. These randomized bases may be of a worse quality than the original ones thus requiring a higher runtime for BKZ. By integrating them in the considerations, the resulting pruning function covers a higher range of possible bases and algorithmic behavior for a dimension.

All benchmark results are registered in an object called `BKZBenchmark` which is unique during the execution and globally readable.

2) Volume of simplexes and number of nodes in the search tree: To estimate the number of nodes to visit in the pruned enumeration n_{nodes}, the volume of a d-dimensional cylinder intersection which is described by $\mathbf{A}$ and lies within a d-dimensional hypersphere with radius A_n has to be calculated. The volume of this sphere is $V_n(A_n) = A_n^n \cdot \frac{\pi^{n/2}}{\Gamma(n/2+1)}$.

To calculate the volume of the cylinder intersection V_{sim}, we employ the method proposed in [12] based on simplexes which is also implemented in the pBKZ-lib [4]. The prerequisites for the pruning function $\mathbf{A}$ are that the dimension of the lattice is even and that it fulfills $A_1 = A_2 \leq A_3 = A_4 \leq \dots \leq A_{n-1} = A_n$. Simplexes determined by $\mathbf{A}$'s of this form are called *even simplexes* and pruning functions describing them *even functions* in the following. An even function also fulfilling $A_i = \frac{\lfloor (i+1)/2 \rfloor}{d/2}$ is called an *even linear function* because of the linear growth between neighboring pairs of values.

The procedure of [12] returns the percentage pr of the volume of the d-dimensional even simplex compared to that of the corresponding d-dimensional hypersphere $V_n(A_n)$. Hence, the volume of the even simplex described by the pruning function $\mathbf{A}$ is: $V_{n,sim}(\mathbf{A}) = pr \cdot V_n(A_n)$. An estimate for the number of the nodes in the search tree n_{nodes}, is the sum for the estimates of nodes for each level of the tree given by

$$n_{nodes} = \frac{1}{2}\sum_{k=1}^{n} \frac{V_{n,sim}(\mathbf{A}_{1:k})}{\prod_{i=n+1-k}^{n} ||\mathbf{b}_i^*||}$$

and $\mathbf{A}_{1:k}$ returns a k-dimensional cylinder intersection given by the first k entries of $\mathbf{A}$ [2], [12].

3) Success probability and overall cost function: The original cost function of [2] $c = \frac{t_{BKZ}+t_{ENUM}}{p}$ is modified to incorporate the parallel execution of several basis reduction-enumeration cycles in parallel. As a simplification, we assume that at each time step all threads finish at about the same time. The chance that a single enumeration succeeds p_{single} is given by the result of the simplex volume calculation. It is the pr-value corresponding to $\mathbf{A}_{1:d}$. The chance p_{par} of j trials in parallel to succeed results from the Bernoulli chain assuming that 'at least one trial is successful' which we calculate by the opposite 'no trial is successful' by $p_{par} = 1-(1-p_{single})^j$.

4) Actual searching algorithm: The search for the pruning function is realized by a parallelized simulating annealing procedure which is summarized in Algorithm 2. An introduction to heuristic optimization with simulated annealing and further details about the algorithm can be found in [13]. A *solution* (sol in Algorithm2) within our simulated annealing represents the combination of a pruning function $\mathbf{A}$ with its corresponding β-configuration. The `costs()` of the solution is the runtime estimated to solve the SVP and the sol_{even_linear} represents the even linear pruning function with the β-configuration that minimizes its costs. Random neighboring solutions as required within `randomModifyToNeighbor()` are generated by first randomly selecting two entries A_j and A_{j+1} with an odd $j \in \{1,\ldots,n-1\}$ from the solutions pruning function $\mathbf{A}$ and randomly increasing or decreasing A_j and A_{j+1} by a percentage $\in [0.0001, 0.03]$ under the constraint that the pruning function remains monotonically increasing.

Algorithm 2: p3Enum-epfg parallel simulated annealing approach.

```
   Input  : $temp_{target}$, $sample\_size$, $max\_iterations$,
            $coolrate \in ]0.0, 1.0[$
   Output : $sol_{min}$
 1 $confid \leftarrow$ max (numberOfThreads(), $\beta$−configurations)
 2 Start parallel execution
 3 for $i \leftarrow 1$; $i \le confid$; $i++$ do
 4     $sol_{cur} \leftarrow$ createStartSolution($sol_{even\_linear}$, i)
 5     $max\_dist \leftarrow$ maxCostDistance($sol_{cur}$, $sample\_size$)
 6     $temp_{init} \leftarrow \frac{max\_dist}{\log(0.8)}$
 7     $temp_{curr} \leftarrow temp_{init}$
 8     $sol_{best} \leftarrow sol_{cur}$
 9     while $temp_{target} < temp_{cur}$ do
10         $sol_{temp} \leftarrow$ randomModifyToNeighbor($sol_{cur}$)
11         for $its \leftarrow 1$; $its < max\_iterations$; $its++$ do
12             if costs($sol_{temp}$) < costs($sol_{curr}$) then
13                 $sol_{curr} \leftarrow sol_{temp}$ if
                     costs($sol_{curr}$) < costs($sol_{best}$) then
14                     $sol_{best} \leftarrow sol_{curr}$
15             else if $e^{-\frac{(\text{costs}(sol_{temp}-\text{costs}(sol_{curr}))}{temp_{curr}}} > rand\ (0,1)$
                 then
16                 $sol_{curr} \leftarrow sol_{temp}$
17         $temp_{cur} \leftarrow coolfac \cdot temp_{cur}$
18 End parallel execution
19 $sol_{min}$ = reduceThreadSolutions()
```

Line 1 determines whether there are more β-configurations in the `BKZBenchmarker` or more available threads. This value sets the number of iterations of the main `for`-loop in line 3 which is executed in parallel.

The function `maxCostDistance()` generates $sample\_size$ random neighbors from sol_{curr} to determine the biggest cost-difference for neighboring solutions, which is required to calculated the initial temperature $temp_{init}$ for the annealing in Line 6 following the literature. $temp_{init}$ is adapted for each iteration of the `for`-loop to optimize the runtime.

To determine an appropriate starting solution, `createStartSolution()` randomizes the sol_{even_linear} in dependence of i. To that end, a β-configuration depending on i is assigned to sol_{even_linear}. When i exceeds the number of β-configurations in `BKZBenchmarker`, the pruning function is additionally randomized by executing $3 \cdot d$ consecutive calls of `randomModifyToNeighbor()` so that each entry A_i is touched six times in average. This procedure assures that each β-configuration is tested at least by one thread, that each available thread is assigned some work, and that the search space is entered from as many points as possible.

Within the parallel loop, the algorithm follows the general simulated annealing procedure and, in particular, allows to randomly accept solutions worse than sol_{curr} in dependence of the current temperature $temp_{curr}$ in Line 15. The left part of Figure 1 shows the development of the first 100 changes in the costs during annealing of the pruning function for 70-dimensional lattice. It shows the increase of the costs between several steps. The right part documents the overall decrease of the costs for the whole annealing process. Finally, the output sol_{min}, which is the best solution of all threads, is returned.

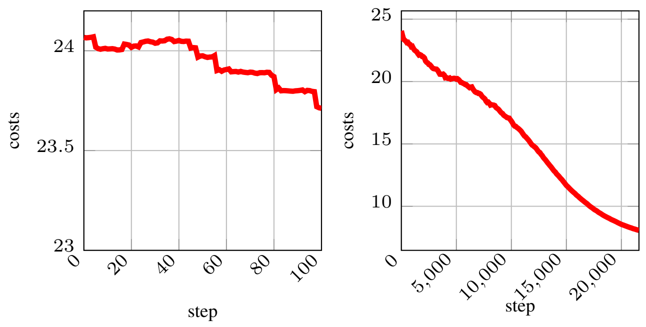

Fig. 1. Development of costs during annealing.

Figure 2 summarizes the software architecture of p3Enum-epfg in UML limited to the important classes, operations and attributes explained in this section.

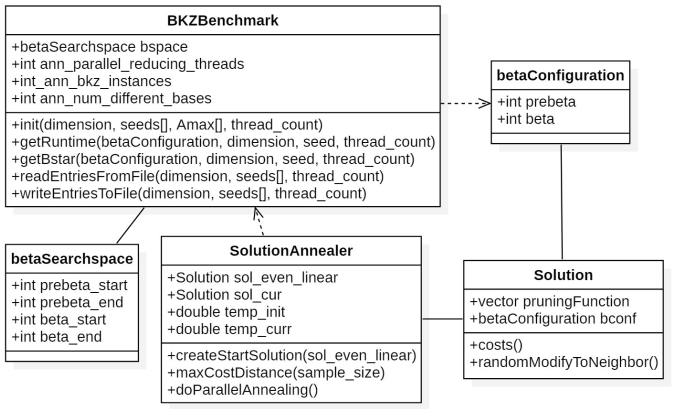
BKZBenchmark
+betaSearchspace bspace
+int ann_parallel_reducing_threads
+int ann_bkz_instances
+int ann_num_different_bases
+init(dimension, seeds[], Amax[], thread_count)
+getRuntime(betaConfiguration, dimension, seed, thread_count)
+getBstar(betaConfiguration, dimension, seed, thread_count)
+readEntriesFromFile(dimension, seeds[], thread_count)
+writeEntriesToFile(dimension, seeds[], thread_count)
betaConfiguration
+int prebeta
+int beta
betaSearchspace
+int prebeta_start
+int prebeta_end
+int beta_start
+int beta_end
SolutionAnnealer
+Solution sol_even_linear
+Solution sol_cur
+double temp_init
+double temp_curr
+createStartSolution(sol_even_linear)
+maxCostDistance(sample_size)
+doParallelAnnealing()
Solution
+vector pruningFunction
+betaConfiguration bconf
+costs()
+randomModifyToNeighbor()

Fig. 2. Software architecture of p3Enum-epfg

V. Evaluation

A. Methodology

The compute nodes employed for the runtime measurements of this paper are dual socket machines with Intel E5-2680 v3 CPUs (24 cores) and 64 GB of RAM. For p3Enum, fplll, and all the libraries they depend on, we use gcc compiler 8.2.0 and for SubSieve gcc 4.9.4.

For lattice dimensions $66-90$, we generated random lattices in the Goldstein-Mayer form of the Darmstadt SVP challenge (D-SVPC) with random seed values (0, 237, 6880, 97575, 98937). The bound for the length $\bar{A}$ is determined by the length of the shortest vector calculated with fplll increased by 2.

For lattice dimensions $92 - 100$ we employed the seeds for which hall of fame entries in the D-SVPC exist and their lengths increased by 2 as $\bar{A}$. Hence, the number of lattices employed per dimension varies from two to four.

For the visualizations of the performance data, we employ box plots of different color. The boxes represent the values between the 25- and 75-percentile, called Q_{25} and Q_{75}, meaning that 50% of the measurement values lie in that range. The lines below and above the boxes represent the so-called whiskers. Their ends indicate the lowest and highest measurement point, respectively, which lies within $1.5 \cdot (Q_{75} - Q_{25})$ of the lower and upper quartile, respectively. The median Q_{50} is shown by the black horizontal lines within the boxes. Outlying measurement points are drawn as circles. Due to the data distribution, no outliers below the whiskers occur.

B. Runtimes with one thread

To establish a performance baseline and to verify the quality of our pruning functions, we compare the runtimes with one thread to the runtimes of fplll. We choose dimension 90 as the highest dimension fplll is able to solve with the provided strategy file and used five lattices with differing random seed. Additionally, we performed a benchmark which measures the number of nodes our program processes per second in the enumeration tree identical to the benchmark in the fplll strategizer. p3Enum achieves $2.6 \cdot 10^7$ nodes per second and fplll $2.99 \cdot 10^7$ nodes per second. Thus, its enumeration is about 15 % faster than p3Enum's. Figure 3 compares the overall runtimes for solving the SVPs on our five random lattices split between the different seeds. For each seed, we performed 20 runs of fplll and 30 runs of p3Enum.

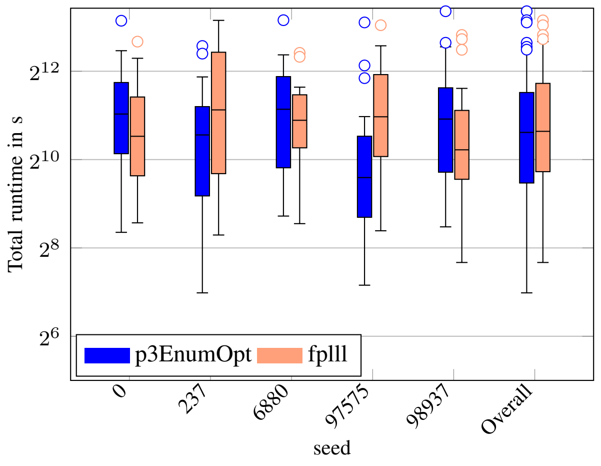

Fig. 3. Comparison for dimension 90 with one thread.

The shape of the runtime distribution for the varying seeds is similar between p3Enum and fplll with an exception for seed 97575 where the boxes have only a small overlapping region. Considering the overall comparison at the right of the diagram, the median of both programs is about the same with 1507 s for p3Enum and 1593 s for fplll which is also true for the average runtime (2199 s vs. 2417 s) where p3Enum is 9 % faster. For all seeds, p3Enum delivers several trials with a lower minimal runtime than fplll, but also for four out of the five seeds at least one run that is slower than all fplll runs. The runtime of p3Enum is slightly faster than the state-of-the-art solution. However, p3Enum's enumeration is slower and internally time-consuming conversions from p3Enum's data structures to fplll's and vice versa have to be performed to employ fplll's BKZ routines as library calls. This indicates that the quality of our pruning functions is better.

C. Parallel runtimes

In this section, we compare the performance of p3Enum employing the new pruning functions and the changed parallelization strategy (cf. Section IV-A) to the former p3Enum-performance, the fplll library, SubSieve and G6K. We split the results into three Figures 4, 5 and 6 since this allows to refine the scale of the y-axis to cope with the exponential growth.

Each box for p3Enum in the new version is created out of 200 single measurement points, while the others are based on at least 75 single measurements. The diamonds in the boxes for p3Enum additionally show the average of all runs for the respective dimension. Finally, the red horizontal lines indicate the average runtime of G6K reported in [8]. Both p3Enum-versions always employ all 24 cores of the test systems.

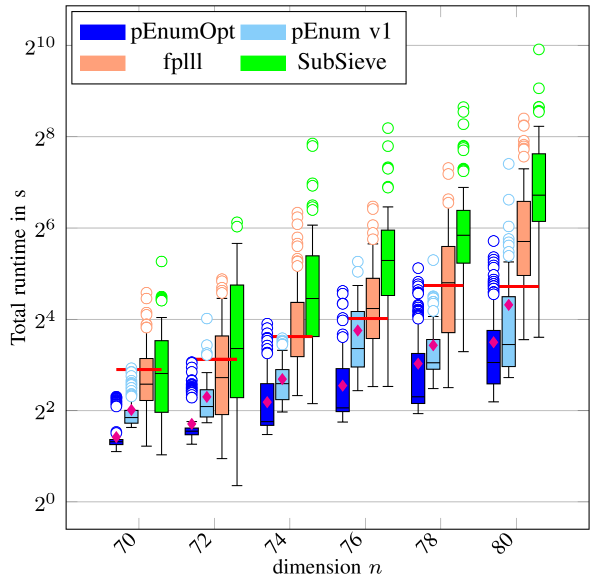

Fig. 4. Performance comparison for dimensions 70-80.

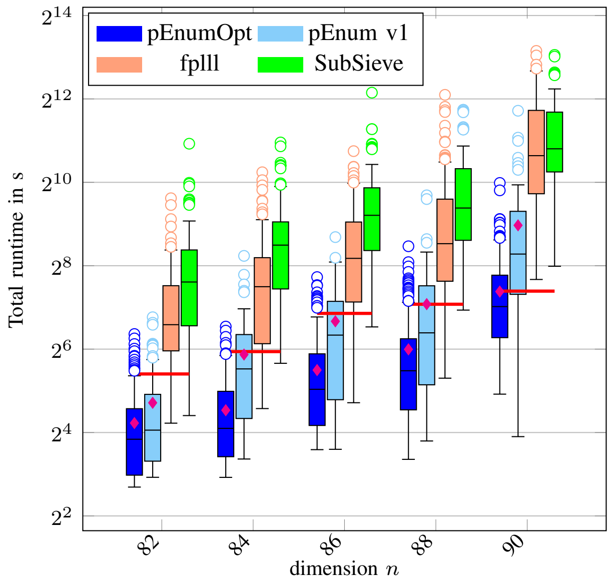

Fig. 5. Performance comparison for dimensions 82-90.

The box plots demonstrate a high variance within the measurement values for all three implementations. Hence, we believe this statistical visualization is very helpful compared to just calculating averages. While pruned enumeration is known to have highly varying runtimes [8], our experiments show that SubSieve also exhibits this characteristic. In short tests with the G6K-code we also reconsigned a similar behavior. We employed the build-in functionality of G6K to solve the exact SVP on our five 90-dimensional lattices which automatically returns the average over the five matrices. We repeated the test 45 times with the fastest run of 20 s and the slowest more than 400 s, a factor higher than 20.

The speedup of the new p3Enum-version for $d \in \{70, \ldots, 80\}$ compared to the original code is a factor between 1.4 and 1.7, while it is between 2.3 to 6.7 times faster than fplll. The speedup compared to SubSieve grows with the dimension from 3.3 at $d = 70$ to 13.3 at $d = 80$. Hence, although the runtime of sieving algorithms is predicted to grow with a lower exponent than the runtime of enumeration algorithms [6], [7] this is not the case for the available implementations and the considered range of lattice dimensions. Furthermore, [8] assumes that a good implementation of the novel sieving algorithms will outperform pruned enumeration already somewhere between $d = 70$ and $d = 80$. The speedup compared to G6K lies between 2.3 and 3.3.

In the range $d \in \{82, 90\}$, the improved p3Enum-version outperforms the original by a factor between 1.4 and 2.1. Compared to SubSieve, the mean value and the average of the runtimes are considerably lower by speedup factors between 14.6 and 19.3. Interestingly, the highest speedup is achieved for $d = 84$ and starts to slightly decrease afterward. Maybe this indicates that the lower growth in complexity of sieving compared to enumeration shows up in this range but p3Enum is still considerably faster. The speedup compared to G6K lies in $[2.2, \ldots, 2.7]$ for $d \in \{82, \ldots, 88\}$ but is just 1.05 for $d = 90$, although the speedup of the new p3Enum-version is the highest for $d = 90$ compared to the old version (factor 3.0) and to fplll (factor 14.5). The matrices for $d = 90$ employed in [8] or the dimension 90 at all seem to be very suitable for G6K's approach. This is underpinned by the fact that p3Enum performs better for dimension 92 again as shown in Figure 6. The speedup compared to G6K is 1.78. From dimension 94 on, where G6K is faster by a factor of 1.13, G6K continuously outperforms p3Enum but its runtimes are still competitive.

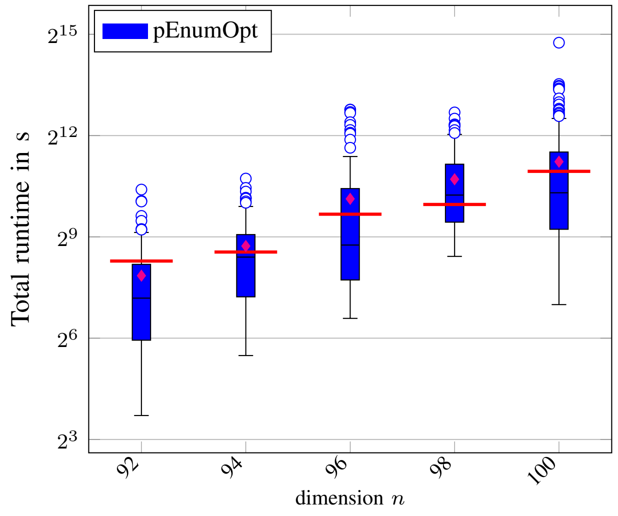

Fig. 6. Performance comparison for dimensions 92-100.

Additionally, Figures 4 and 5 highlight another important advantage of p3Enum: The parallel execution has a stabilizing effect on the runtimes for all dimensions. For example, the range of highest and lowest runtime for dimension 74 is 75 s

for fplll but only $10\,s$ for p3Enum. This allows a much better reproducibility and predictability of the runtimes and hence makes p3Enum more practical and usable in applications where an a priori knowledge of the expected runtimes is important. The logarithmic scale of the y-axis somewhat hides that this statement is also true for higher dimensions. While the range of runtimes for dimension 90 is $882\,s$ for p3Enum, it is $8884\,s$ for fplll.

D. Scaling behavior

To investigate the scaling behavior of the new p3Enum-version, we take the SVP for the 88-dimensional lattice with seed 0 as an example. We consider the overall runtimes to solve the SVP when setting the number of threads to 1, 2, 4, 8, 16 and 24. For each number of threads, sixty runs are taken into account resulting in 360 measurement points. The results are summarized by box plots in Figure 7. The red line connects the averages of the runtimes while the red numbers show the relative speedup in the average runtime.

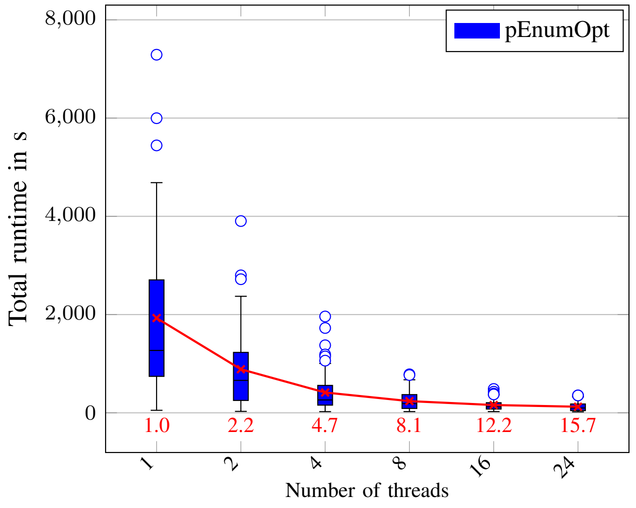

Fig. 7. Scaling behavior of the optimized p3Enum-version.

For a fixed number of threads, there is again a high variance in the runtimes resulting from the randomness within the algorithm itself. The speedup in the average runtime is ideal or even super-linear for 2, 4 and 8 threads. Afterward, the performance of the increasing number of parallel instances of BKZ and ENUM is limited by the memory bandwidth of the test system. In particular, this means that the number of nodes in the search trees which is processed per second within the different threads decreases by up to 30 %. Since the efficiency for 24 threads is still higher than 0.67 a moderate increase of the number of threads will further noticeably improve the runtimes.

Besides the overall runtimes we can see a second important effect of the number of threads which is the span between the fastest and slowest attempt. While the runtimes differ for slightly more than $7200\,s$ for a single thread they only differ by about $320\,s$ for 24 threads limiting the range by a factor higher than 22 compared to a single thread. This considerably increased stability of the runtimes is very important for an SVP solver to be practical and further increasing the number of threads employed will further increase the stability of the runtimes.

E. Pruning functions

Figure 8 compares different pruning functions for a lattice of rank 90. The light blue line shows the interpolated line resulting from the procedure of [3] which was employed in the former p3Enum version. The red line shows the pruning function which is taken from the default strategy file of the fplll library. The dark blue line represents the function which results from the simulated annealing while the dashed line is the even linear starting point of the optimization process. All three pruning functions depicted have in common that they increase the y-values in the very first part of the graph compared to the even linear function, then fall below it at different entries A_i of $\mathbf{A}$. The crossover point varies between $i = 13$ for the polynomial and $i = 20$ for fplll. Interestingly, all consistently intersect the even linear graph around entry 58 of $\mathbf{A}$ again and assume values higher than the starting function from there on. The lower the y-value the more aggressive the pruning function at this position, meaning that the polynomial from [3] cuts of many more vectors in the range $1 \leq i \leq 3$ than the other two while the function from simulated annealing accepts much more candidates.

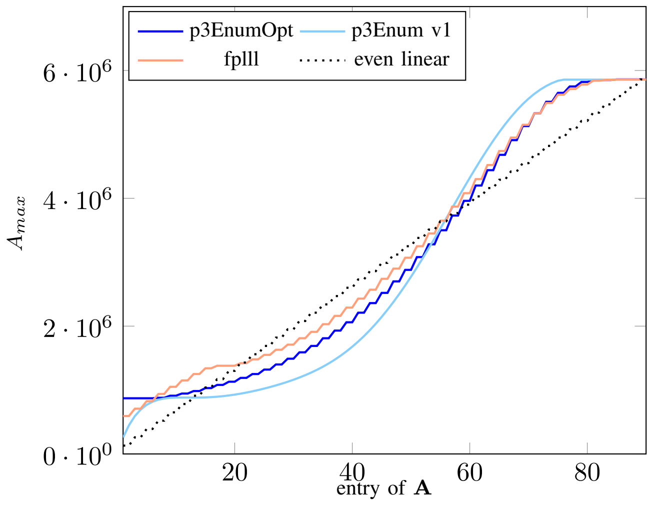

Fig. 8. Different pruning functions for lattices with $d = 90$.

The function of [3] is smooth resulting from the fact that it is represented by a polynomial of degree eight, while the other two have the shape of stairs since they are based on even simplexes. Section V-B shows that the usage of our function performs comparably to fplll meaning that different pruning functions can result in the same effective runtime of the solution process.

Additionally, we noticed that besides all randomizations within the simulated annealing two independent runs to create a pruning function for dimension 90 with 24 threads converged to nearly the same $\mathbf{A}$. The maximum deviation between two corresponding A_is is $0.2\,\%$ while the average deviation over all entries is only $0.035\,\%$. The resulting cost is $1704.90\,s$ and $1704.89\,s$, respectively, indicating that a nearly ideal pruning function was found.

VI. Conclusion and Outlook

We presented two important extensions to the SVP solver p3Enum. A novel parallelization for the enumeration with extreme pruning and optimized pruning functions generated with the new p3Enum-epfg module by a shared memory parallelized simulated annealing approach speed up the former p3Enum-version by a factor up to 3. This makes p3Enum the fastest SVP solver in the range of $d = 66$ to $d = 92$ which makes p3Enum a good SVP-oracle in lattice reductions frameworks when higher values of β are required. Additionally, we performed a detailed study of state-of-the art enumeration- and sieving-based SVP solvers and demonstrated the randomness within all algorithms considered, including the sieving-based SVP solvers. This shows that only considering median or average values may be misleading when evaluating the tractability. The parallelization makes p3Enum's runtimes more stable and reproducible also in comparison to sieving-based SVP solvers and employing more threads will further stabilize and reduce the runtimes.

In the future, we will increase the serial performance of the enumeration to be at least the same as fplll's which enables a more detailed comparison of the pruning functions generated by simulated annealing. We also will consider random Goldstein-Mayer lattices of higher dimension and other types of lattices like those resulting from knapsack problems. Furthermore, we will extend our studies of the convergence behavior of simulated annealing and optimize it's parameterization.

Acknowledgments

This work has been co-funded by the DFG through CRC 1119 CROSSING and BI 714/6-1. Calculations were conducted on the Lichtenberg high performance computer of the TU Darmstadt, and computing resources granted by RWTH Aachen University under project prep0016. We thank Yuntao Wang for helpful discussions about progressive BKZ and the pBKZ-lib code. We also thank Christian Iwainsky for the discussions and ideas on parallelization on different levels of the algorithm and the anonymous reviewers for their helpful comments.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172

[2] N. Gama, P. Q. Nguyen, and O. Regev, "Lattice enumeration using extreme pruning," in *Advances in Cryptology - EUROCRYPT 2010*, H. Gilbert, Ed. Springer, 2010, pp. 257–278.

[3] P.-C. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. Buchmann, C.-M. Cheng, and B.-Y. Yang, "Extreme enumeration on GPU and in clouds," in *Cryptographic Hardware and Embedded Systems - CHES 2011*, B. Preneel and T. Takagi, Eds. Springer, 2011, pp. 176–191.

[4] Y. Aono, Y. Wang, T. Hayashi, and T. Takagi, "Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator," in *Advances in Cryptology - EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Springer, 2016, pp. 789–819.

[5] M. Burger, C. Bischof, and J. Krämer, "p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver," in *International Conference on Computational Science – ICCS-2019*, J. Rodrigues, P. Cardoso, and R. e. a. Lam, Eds. Springer, 2019.

[6] T. Laarhoven and A. Mariano, "Progressive lattice sieving," in *Post-Quantum Cryptography - PQCrypto 2018*, 2018, pp. 292–311. [Online]. Available: https://doi.org/10.1007/978-3-319-79063-3_14

[7] L. Ducas, "Shortest vector from lattice sieving: A few dimensions for free," in *Advances in Cryptology - EUROCRYPT 2018*, 2018, pp. 125–145. [Online]. Available: https://doi.org/10.1007/978-3-319-78381-9_5

[8] M. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E. W. Postlethwaite, and M. Stevens, "The general sieve kernel and new records in lattice reduction," to appear, 2019.

[9] C. P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Mathematical Programming*, vol. 66, no. 1, pp. 181–199, Aug 1994. [Online]. Available: https://doi.org/10.1007/BF01581144

[10] T. f. development team, "fplll, a lattice reduction library," 2016, available at https://github.com/fplll/fplll. [Online]. Available: https://github.com/fplll/fplll

[11] J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel, "Parallel shortest lattice vector enumeration on graphics cards," in *Progress in Cryptology - AFRICACRYPT 2010*, D. J. Bernstein and T. Lange, Eds. Springer, 2010, pp. 52–68.

[12] Y. Aono, "A faster method for computing gama-nguyen-regev's extreme pruning coefficients," *CoRR*, vol. abs/1406.0342, 2014. [Online]. Available: http://arxiv.org/abs/1406.0342

[13] P. Van Laarhoven and E. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.