2018 IEEE International Conference on Computational Science and Engineering

Exploring the Performance Envelope of the LLL Algorithm

Michael Burger*, Christian Bischof*, Alexandru Calotoiu†, Thomas Wunderer‡ and Felix Wolf†
*Scientific Computing
TU Darmstadt, 64283 Darmstadt
Email: {michael.burger, christian.bischof}@sc.tu-darmstadt.de
†Parallel Programming
TU Darmstadt, 64283 Darmstadt
Email: {calotoiu, wolf}@cs.tu-darmstadt.de
‡Cryptography and Computer Algebra
TU Darmstadt, 64283 Darmstadt
Email: wunderer@cdc.tu-darmstadt.de

***Abstract*—In this paper, we investigate two implementations of the LLL lattice basis reduction algorithm in the popular NTL and fplll libraries, which helps to assess the security of lattice-based cryptographic schemes. The work has two main contributions:**

First, we present a novel method to develop performance models and use the unpredictability of LLL's behavior in dependence of the structure of the input lattice as an illustrative example. The model generation approach is based on profiled training measurements of the code and the final runtime performance models are constructed by an extended version of the open source tool Extra-P by systematic consideration of a variety of hypothesis functions via shared-memory parallelized simulated annealing. We employ three kinds of lattice bases for our tests: Random lattice bases of Goldstein-Mayer form with linear and quadratic increase in the bit length of their entries and NTRU-like matrices. The performance models derived show a very good fit to the experimental data and a high variety in their range of complexity which we compare to predictions by theoretical upper bounds and previous average-case estimates. The modeling principles demonstrated by the example of the use case LLL are directly applicable to other algorithms in cryptography and general serial and parallel algorithms.

Second, we also evaluate the common approach of estimating the runtime on the basis of the number of floating point operations or bit operations executed within an algorithm and combining them with theoretical assumptions about the executing processor (clock rate, operations per tick). Our experiments show that this approach leads to unreliable estimates for the runtime.

***Index Terms*—performance models, model fitting, parallel simulated annealing, OpenMP**

I. Introduction

To assess the security of a cryptographic scheme which is based on some computational problem one has to estimate the hardness of the underlying problem. To that end, cryptanalysists aim at estimating the complexity of known algorithms or attacks to solve these problems. Since a real execution of such algorithms on secure instances of cryptographic interest can not finish in acceptable time (otherwise the corresponding cryptosystem could be broken), models for the runtime depending on the inputs of these algorithms are required. These models, called performance models in the following, can then be used to choose cryptographic parameters providing a certain security level. However, if the generated models overestimate the runtime of an attack the resulting encryption may be insecure. Underestimating the runtime of attacks may result in selecting unnecessary large cryptographic parameters and hence in an inefficient cryptosystem. In this paper, we present a novel method to generate accurate performance models based on real measurement data. As a use case we decided for the LLL basis reduction algorithm [1], named after its inventors Lenstra, Lenstra and Lovász. It is a common building block in the emerging field of lattice-based cryptography and its runtime depends on several factors like the actual variant of the LLL procedure, the type of basis to reduce, the parametrization of the algorithm or the precision of the floating point (FP) variables employed. Stehlé [2], e.g., states that the practical behavior of LLL is considered as mysterious, while Lin [3] calls the variable complexity of LLL a drawback of the algorithm. This unpredictability of the behavior of LLL makes it a good candidate to highlight the advantages of our method.

In this work, we propose a new approach to model the complexity of algorithms by the example of the LLL algorithm. We extend the open source software Extra-P[1] with a shared-memory parallelized variant of the simulated annealing [4] heuristic, thus exploring a wide range of fitting functions for both runtime and floating point operations executed in parallel requiring only little time. We develop performance models for two LLL implementations from the NTL[2] and the fplll[3] libraries. Both are widely used in the cryptographic community. With three different types of lattices, we demonstrate the quality and accuracy of the runtime predictions resulting from

Work supported by German Science Foundation (DFG) through SFB 1119 and WO 1589/7-1. Calculations were conducted on Lichtenberg HPC system of TU Darmstadt.

[1] http://www.scalasca.org/software/extra-p/download.html
[2] http://www.shoup.net/ntl/
[3] https://github.com/fplll/fplll

978-1-5386-7649-3/18/$31.00 ©2018 IEEE
DOI 10.1109/CSE.2018.00012

our generated models.

II. Preliminaries and an overview of LLL

A lattice is a discrete additive subgroup of $\mathbb{R}^n$. In this case, n is called the dimension of the lattice. For each n-dimensional lattice $\Lambda \neq \{\mathbf{0}\}$ there exist $d \leq n$ linearly independent vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ such that $\Lambda = \mathbb{Z}\mathbf{b}_1 + \ldots + \mathbb{Z}\mathbf{b}_d$. Such a set of vectors is called a basis of Λ. The cardinality d of a basis is uniquely determined by the lattice and called the rank of the lattice. A full rank lattice is a lattice with $n = d$.

The LLL algorithm [5] is one of the most famous lattice basis reduction algorithms. Basis reduction is the process of improving the "quality" of a lattice basis and can be used to find short vectors in lattices. At the time of its invention LLL was the first polynomial-time basis reduction algorithm. However, the short vectors found with LLL are typically exponentially larger than the shortest non-zero vectors in the lattice. LLL proved to be useful for a variety of applications like factoring integers and polynomials, linear programming or lattice-based cryptography. In particular, other basis reduction algorithms such as BKZ [1] and its variants like BKZ 2.0 use LLL extensively. All competitive LLL libraries are based on floating point arithmetic [2]. Theoretical upper bounds on the runtime of floating point LLL are well-established but the algorithm is known to behave much better in practice than what can be deduced by these upper bounds [6], [7].

Algorithm 1 provides a high level pseudocode of the LLL algorithm hiding the details of practical improvements like proposed, e.g., in [1], [6]. For detailed pseudocode of several FP implementations of LLL refer to [2]. δ determines the quality of the returned basis. Smaller values improve the quality but increase the runtime.

Algorithm 1 The LLL Algorithm

Input: A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ of a lattice Λ and $\delta \in [\frac{1}{4}, 1]$
Output: Basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ is LLL-reduced with factor δ

1: Compute all the Gram-Schmidt coefficients $\mu_{i,j}$ and the lengths $||\mathbf{b}_i^*||$
2: **for** $i = 2$ to d **do**
3: Size-reduction of basis vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$ in reverse order
4: Update Gram-Schmidt values where required
5: **end for**
6: **if** Condition based on δ is not satisfied for some vectors $\mathbf{b}_j$ and $\mathbf{b}_{j+1}$ **then**
7: swap $\mathbf{b}_j$ and $\mathbf{b}_{j+1}$, return to Line 1
8: **end if**

III. Floating point LLL and performance models

Several variants were proposed after Schnorr's first provable correct floating point algorithm in 1986 [8]. It is bound by $O(d^3 n(d + \log B)^2 \log B)$ where $B = \max_i\{\|\mathbf{b}_1\|, \ldots, \|\mathbf{b}_d\|\}$ and thus $\log_2 B$ is the bit size of the input matrix entries. One variant was proposed in 2009 by Nguyen and Stehlé [6] whose upper bound is $O(d^4 n \log_2 B(d + \log_2 B))$ and hence only grows quadratically in the bit size of the entries. This approach was further optimized in 2011 by Novocin et al. [9]. They presented a variant of LLL, called $\widetilde{L}^1$, whose runtime depends only quasi-linear on $\log_2 B$ and is still polynomial in d. This variant is mainly relevant in the case of large $\log_2 B$ values. Its upper bound complexity is $O(d^{5+\epsilon} \log B + d^{\omega+1+\epsilon} \log^{1+\epsilon} B)$ where ω is a valid exponent for matrix multiplication and $\epsilon > 0$. Stehlé [2] states that for floating point LLL like the provable algorithm of Schnorr [8] or L^2 [6] we have a complexity of total degree 7 but that the complexity bound of L^2 is always better than that of [8].

Nguyen et al. [7] employed the implementation in the fplll library to investigate floating point LLL and determined models for its average runtime. They employ two different kinds of input bases, Ajtai-type random bases and Knapsack-type bases, and report an average complexity of $O(d^4 \log^2 B)$ hence, one degree lower than the theoretical bound.

Ling et al. [3] investigated the complexity of special variants of LLL for problems arising in the field of communication and signal processing. They demonstrate that the average complexity that is expected is much lower than the worst-case bounds. They consider the number of FLOPs for LLL and derive an average complexity bound of $O(n^3 \log n)$ for some of their special problem instances and the resulting lattices. They highlight that LLL's complexity is highly dependent, i.e,. the exponent of n changes, with the basis to be reduced.

Automatically generated empirical performance models were used to identify scalability issues in HPC codes by Calotoiu et al. [10]. This performance modeling approach was successfully employed by Iwainsky et al. [11] to investigate the scalability of various OpenMP implementations on different architectures with a high number of SMP cores.

IV. Modeling approach based on Extra-P and parallel simulated annealing

Extra-P [10] was originally created as a tool to assist in identifying scalability bugs in parallel programs. To this end, functions are generated from data describing the relation of the runtime of different code regions with different variables like the size of the input data. Calotoiu et al. [10] showed that the runtime complexity of many code regions and algorithms can be modeled by functions of the type

$$\sum_{k=0}^{m} c_k \cdot x^{i_k} \cdot \log_2^{j_k}(x) \tag{1}$$

where x is the problem size for the experiments of this paper. But it can also be the number of threads employed or some target precision of the result. Functions of the shape of Equation 1 are called to be in performance model normal form (PMNF). Reisert et al. [12] showed that in most cases $m = 1$ is sufficient to achieve a good result and, in particular, that those models are even the best ones.

As it turns out, also in our application, always the model with $m = 1$ results in the best prediction. Hence, all our models have the shape $c_0 + c \cdot x^i \cdot \log_2^j(x)$.

To determine appropriate values of the parameters c_0, c, i and j, we extended Extra-P by a novel model generator to systematically explore the space of all possible model functions. First, so called *training data* is required, represented by profiled runtime measurements with a varying parameter (see also Section V-C). This parameter is, in our case, the rank of the lattice d, which is equal to the dimension n. We also generate a first model with Extra-P's default model generator.

The training data and the initial model are the inputs of our extension which employes the simulated annealing heuristic [4] to further improve the model. The internal energy, representing the quality of the model, is the Residual Sum of Squares (RSS) which is defined as:

$$RSS = \sum_{i=0}^{n} (y_i - f(x_i))^2 \tag{2}$$

There, i is the number of the measurement y_i and $f(x_i)$ is the evaluation of the model function corresponding to the x value of the same measurement. If $RSS = 0$, then $y_i = f_i \; \forall i$ and all points of the training data lie directly on the fitting curve, which is the ideal case. The simulated annealing procedure is shared-memory parallelized with OpenMP. As initialization, the starting point for each thread is randomly set around the initial solution delivered by Extra-P, i.e., the coefficients i and j are randomly modified by each thread. Then, each thread performs its own search for the best solution until the temperature cooled down to the target. We empirically determined that $T_{new} = T_{old} * 0.999$ is a good choice for the cooling function and that $T = 0.00001$ is suitable stopping criterion. Our software automatically generates a report file in LaTeX which allows to easily compare the solutions of the different threads, their quality and the consistency of the result. It visualizes the fitting of the training data and the generated model as well as the prediction for higher values of the rank d. In contrast to Extra-P this avoids runtime intensive exhaustive model search or time-consuming manual trial-and-error adaption of i and j to find the best model. The software is available for download[4].

Extra-P is not only capable of modeling the development of the compute time, but it can also generate models for recorded hardware performance counters, like the number of memory accesses or the number of floating point operations (FLOPs) executed depending on the size of the input parameter. This enables us to also compare theoretical estimates on the number of FLOPs in LLL with the actual number.

V. Methodology

In this section, we describe the LLL implementations used, discuss how we created the required training data for our models and the input bases for our tests.

[4]https://github.com/MiBu84/SMP-Simulated-Annealing

A. Employed LLL implementations and parameters

In the NTL library, we investigate the LLL_RR routine, which works with an arbitrary floating point precision data type. The implementation is based on the description from [13] that already includes improvements like incremental and on-demand calculation of Gram-Schmidt coefficients. Furthermore, the NTL library adds several heuristics concerning the acceptable precision loss and the number of type conversions. Additionally, we considered the implementation of the L^2 algorithm [14] in the fpLLL library.

The runtime of FP LLL implementations depends besides the input basis and the chosen δ-value on the precision of the employed floating-point type. Smaller dimensions can theoretically be calculated with a lower precision. The required precision for a dimension should be predicted in advance and set to the smallest possible value. Nguyen et al. [7] state, e.g., that for their LLL implementation the FP precision can be estimated by $0.18d + o(d)$ in the average case. To simplify our experiments, we set the FP precision to 150 bits since this value is sufficient to run our highest tested dimensions. We always set $\delta = 0.999$.

B. Test systems

We employ two different systems for our tests, summarized in Table I. All measurements are from single-thread runs. The compute thread is bound to the second physical core and the systems are used exclusively. We use NTL in version 10.0.5 and fplll in version 5.2.0.

Table I: Test systems with different Intel Xeon processors

	Westmere	**Haswell**
CPU	8*E7-8837	2*E5-2680 v3
Cores	8c@2.66 GHz	24c@2.50 GHz
RAM	1024 GB	64 GB
OS	CentOS 7.3	CentOS 7.3

C. Generating training data

The LLL_RR from NTL was uniformly called with LLL_RR(L,0.999,0,0,1), where the second parameter represents δ, within a C++ driver application where L is the input lattice read and stored as NTL mat_ZZ. fplll was used as executable and we set $\delta = 0.999$ via command line argument.

To generate the models, we instrumented the LLL-functions with Score-P [15] which is a measurement infrastructure specifically targeted to parallelized codes but also suitable for serial codes like our underlying LLL implementations. We ensured by non-instrumented runs that the runtime of LLL was not considerably influenced by the instrumentation. Three runs for each rank d of the lattice were performed since the runtimes are very stable for all d's. The rank d was increased in steps of five, while the type of input lattice determines the largest rank tested. Each set of three runs with the same d is called a *measurement point* in the following and the part of all measurement points which is employed for the model generation is our *training data*. Runs with higher rank than the training data are employed to evaluate the quality of the prediction resulting from the generated model.

In addition to the runtime, we consider the number of floating point operations that are executed in LLL_RR. To this end, we employed hardware performance counters through the PAPI interface [16] and logged the value of the PAPI_FP_OPS preset event. This provides an estimate of the total number of operations during the execution of the instrumented code region. We executed those tests on older Intel Westmere processors since counting the floating point operations on Sandy and Ivy Bridge processors may lead to unreliable results[5].

D. Employed lattice types

We employ three types of full rank lattices for our experiments. First, we use lattices resulting from the generator of the Darmstadt SVP challenge[6].

The bases generated are of Goldstein-Mayer form [17], which is shown in Equation 3. The first highlighted column consists of large integers fulfilling the prerequisites listed in [17] and they have a length of $10d$ bits. The diagonal elements are set to 1 while the rest contains zeros.

$$\begin{pmatrix} k_1 & & & \\ k_2 & 1 & & \\ \vdots & & \ddots & \\ k_d & & & 1 \end{pmatrix} \tag{3}$$

One property of those bases is that $\log B = O(d)$, i.e., we can neglect $\log B$ during our model generation, since there is a correlation between $\log B$ and d. We empirically determined that for our bases $\log B = \alpha \cdot d,\ \alpha \in [9.986, 9.999]$. Consequently, modeling the runtime as a function of d is valid and $O(d^4 \log^2 B)$ can be simplified to $O(d^4 \cdot d^2) = O(d^6)$.

The second type of lattices are random lattices with a bit length of d^2 for the k_i's. We modified the code of the Darmstadt SVP challenge generator to enable the creation of such bases. Again, we verified the generated bases and solved $\log B = d^\gamma$ for γ that is always in $[1.999, 2]$. Thus, $O(d^4 \log^2 B)$ is simplified to $O(d^4 \cdot d^{2^2}) = O(d^8)$.

The third type are NTRU-like bases, generated by the latticegen utility of fplll. The length of the first sampled integer q is set to the fixed size of d^2 bits. Equation 4 shows the general shape of these matrices of rank $d = 2N \times 2N$. Details can be found in the fplll documentation[7].

$$\left(\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{N-1} \\ 0 & 1 & \cdots & 0 & h_{N-1} & h_0 & \cdots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & h_1 & h_2 & \cdots & h_0 \\ \hline 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & q \end{array}\right) \tag{4}$$

We also ran some tests with knapsack-like matrices from the fplll generator leading to equivalent model quality as for the three types described above. We are confident that the matrix types considered are representative for other types as well and meaningful for our experiments.

[5] http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops
[6] https://www.latticechallenge.org/svp-challenge/
[7] https://github.com/fplll/fplll/blob/master/README.md\#NS09

VI. Evaluation

First, we consider the models that allow to predict the runtime of both LLL implementations depending on the type of the input matrix. Then, we discuss the common approach to predict the runtime of algorithms based on a model on FLOPs or bit operations required during the execution.

A. Runtime prediction

We start with the two types of random lattices followed by NTRU-like matrices.

1) Random lattices with bit length $10d$

In Figure 1a, we show our training data (green diamonds), the model generated by our extended Extra-P version (blue line) and additional measurements for higher dimensions (red circles) for the LLL_RR routine of the NTL library. Furthermore, the brown curve shows the development of the average model taken from [7]. For the brown curve, we adjusted the coefficients to fit the training data best and adapted the exponents, i.e. d^6 and $\log^0 d$. Hence, this is a kind of best-case scenario for the average model. A comparison of our models and the average model from [7] on a logarithmic scale is shown in Figure 5.

Figure 1a demonstrates that our modeling function fits the set of training data very well. The zoom in the figure highlights this fact since the centers of all green diamonds lie on the blue graph. The model that relates the problem size d to the runtime t_{LLL_RR} in seconds is

$$t_{LLL\_RR}(d) = -0.05 + 8.31 \cdot 10^{-8} \cdot d^{3.05} \cdot \log_2^{3.49}(d) \tag{5}$$

The interesting part is $d^{3.05} \cdot \log_2^{3.49}(d)$, which grows much slower than the upper bound of $O(d^5 \cdot \log^2 B)$ [7] that can be simplified to $O(d^7)$ for our bases. The small negative coefficient results from the short runtimes for small values of d. For example, $d = 50$ requires only slightly more than one second resulting in small jitter in the model. For an ideal model this coefficient would be positive.

The visual impression of the very good fit of the model to the training data is underlined by the quality metrics. The RSS is 1.95 while the highest occurring value in the training data set is 63.

We also compare the predictions of the model to the real runtimes for higher dimensions. We increase the size of the lattice to $d = 290$, i.e., we increase the problem size by a factor of three compared to the training data. The blue line is a good prediction for the red measurement points. For ranks $d \in [100, 225]$, the measurements lie on the line or very near to it, while for higher dimensions there is a small gap. For the last point at $d = 290$, the relative difference between the prediction (4125 seconds) and the actual measurement (3133 seconds) is 24%.

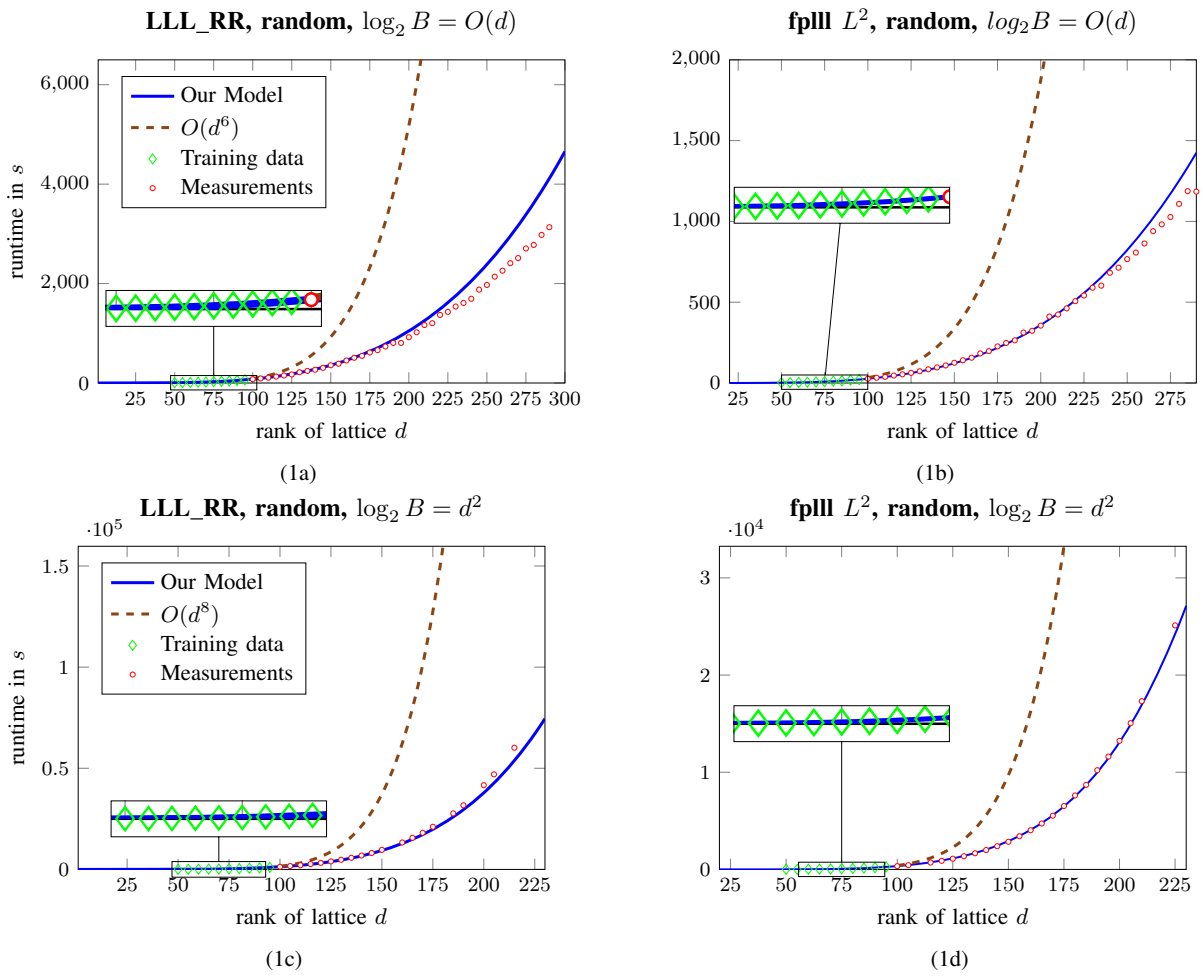

Figure 1: Top diagrams: Extrapolation of runtime for linear growth in bit length. Bottom: Case of quadratic growth.

The same procedure was applied to the fplll library and Figure 1b summarizes the results. The behavior is very similar to the NTL library. Again, the model generated fits the training data very well. The comparison of the RSS value 0.15 to the highest value of 21.05 in the training data underlines this visual impression.

The graph of the model is described by

$$t_{fplll}(d) = -0.21 + 2.51 \cdot 10^{-8} \cdot d^{3.07} \cdot \log_2^{3.5}(d) \quad (6)$$

So, the exponents are about the same as those of LLL_RR (see Equation 5) and are considerably smaller than the average-case estimate $O(d^4 \cdot \log^2 B)$ of Nguyen and Stehlé [7]. For higher ranks, the prediction also overestimates the real values (about 17%), as shown on the right side of Figure 1b.

We notice that although we consider a modified floating point LLL variant (L^2 instead of L^3) and a different library (fplll instead of NTL), the generated model has the same shape and a comparable accuracy as for the NTL implementation for this type of input lattices. However, the fplll library finishes in about a third of the time.

2) Random lattices with bit length d^2

Here, we have a look at random lattices in which the bit length of the entries grows quadratically in the rank d of the lattice, hence $\log_2 B = d^2$, resulting in an average complexity of d^8 (see Section V-D). Equation 7 shows the derived model function for the NTL library.

$$t_{LLL\_RR}(d) = 14.10 + 2.01 * 10^{-8} \cdot d^{4.38} \cdot \log_2^{2.50}(d) \quad (7)$$

The complexity is significantly higher than for bit length $10d$ but still much lower than theoretical bounds. The corresponding model graph with training data and further data points is visualized in Figure 1c. The fit of graph and training data is good. RSS is 100 with 983 being the highest training point. Measurements were performed up to $d = 225$. For higher ranks, the lattice generator is too slow and cannot create a random base for $d = 230$ in more than 50 hours. However, as

far as we can create measurements, we see a strong agreement of the predictions and the actual measurements. The actual value at $d = 215$ is underestimated by only 9%.

Our model for fplll and random lattices with bit length d^2 is described by Equation 8.

$$t_{fplll}(d) = 0.13 + 9.72 \cdot 10^{-10} \cdot d^{4.75} \cdot \log_2^{2.49}(d) \quad (8)$$

The fit of training data and model is nearly ideal since RSS is only 2.75 and the relative difference at $d = 225$ between measurement and prediction is smaller than 4%. In general, all measurements lie directly on the graph. Hence, our modeling approach delivers very accurate models for this type of lattices, while common estimates differ for several orders of magnitude. Additionally, both libraries show a comparable complexity although their underlying LLL variants differ.

3) NTRU-like lattices

Finally, we have a look at the behavior of the LLL implementations when applied to the NTRU-like matrices. Equation 9 gives the model function for the NTL library.

$$t_{LLL\_RR}(d) = -0.90 + 9.20 \cdot 10^{-8} \cdot d^{4.40} \cdot \log_2^{2.5}(d) \quad (9)$$

The exponents in Equation 9 are the same as in Equation 7 for random lattices but the coefficients differ, resulting in a considerably higher runtime of a factor higher than 6 for $d = 135$. This demonstrates that even an accurate knowledge of the complexity class is not sufficient for runtime predictions but that those predictions are possible within our approach.

Figure 2a shows that like in the other examples the model fits the training data (RSS=19.00) very well. Because of the high runtimes, the training data has been generated for $d \in [30, 70]$ and the highest measurement point is at $d = 135$. There, the model underestimates the actual runtime by about 28% indicating that the behavior for small ranks seems to slightly differ from those of higher ones.

Equation 10 shows the corresponding model for the fplll library and the NTRU-like matrices. In that case, we see for the first time a considerable difference in the exponents between fplll and NTL.

$$t_{fplll}(d) = -0.19 + 7.68 \cdot 10^{-8} \cdot d^{5.25} \cdot \log_2^{0.00}(d) \quad (10)$$

As Figure 2b shows, the model nearly ideally fits the training data. The prediction for the higher ranks is also very good. At $d = 160$ the model underestimates the real value by less than 9%, so we again have an accurate model.

B. Predictions based on FLOPs

Nguyen and Stehlé [7] also estimated the number of floating point operations within LLL during their derivation of the overall complexity. In general, it is common to determine the complexity in terms of executed floating point or bit operations. They state that there are $O(d^2 \cdot \log B)$ swaps during the LLL execution and each swap results in an iteration of a loop that contains $O(dn)$ arithmetic operations. Consequently, for a full rank lattice, the number of FLOPs has an upper bound of $O(d^4 \cdot \log B)$. Given the random lattices from the Darmstadt SVP challenge lattice generator, the number of FLOPs is then bounded by $O(d^5)$. In this section, we try to find an appropriate model for the range of ranks d from 50 to 290. We conducted the analysis with LLL_RR since [7] also considers in principle L^3 and not L^2.

Figure 3 shows the model resulting from the training data for $d = 50$ to $d = 100$ as well as additional measurements. The function of the graph is:

$$f_{LLL-RR}(d) = 1.21e^6 + 1.0 \cdot d^{2.55} \cdot \log_2^{3.48}(d). \quad (11)$$

We see that Extra-P predicts a FLOPs increase of $O(d^{2.58} \cdot \log_2^{3.48})$ in contrast to the theoretical upper bound of $O(d^5)$. Consequently, the expected average number of FLOPs is much lower than the theoretical bound.

Now, we investigate the common assumption that the runtime of a program can be predicted by modeling or calculating the required number of floating point operations and combining these results with assumptions about the target system. For the Intel processor employed, we can in general assume one floating point operation per cycle [18]. However, modern vector units like SSE4 in the case of Westmere and AVX/AVX2 in more recent processor generations can increase the throughput to up to 16 FLOPs per cycle when using fused multipy-add-operations (FMA) for AVX2. In our case, the vector units are not employed efficiently since the Westmere and Haswell processors run equally fast and an investigation of the assembly code underpins this impression. For an upper bound on the runtime we assume that only one FP-operation is performed at once. Hence, the relation of the number of FLOPs f, the clock rate of the CPU CLK and the runtime t can be considered as:

$$t = \frac{f}{CLK} \quad (12)$$

Figure 4 shows the resulting runtime prediction compared to the measured runtimes. The distance between both curves considerably increases from small ranks to larger ones and for $d = 290$ the red curve underestimates the real values by a factor of more than 4.5. This difference would further increase for higher ranks since the runtime of the algorithm on the Westmere system is modeled by $t_{LLL\_RR}(d) = -0.277 + 1.526 \cdot 10^{-7} \cdot d^{3.5} \cdot \log_2^{1.5}(d)$ and hence increases faster than the actual number of FLOPs.

There are several factors that contribute to this divergence. The actual performance of software on today's sophisticated computing platforms depends on more factors than FLOPs. For example, memory accesses and their patterns, and resulting cache misses or alignment issues have a significant influence on the actual performance. In the underlying LLL application, we have another important factor. We are not dealing with standard single- and double-precision values, but NTL and fplll work with arbitrary precision floating-point arithmetic.

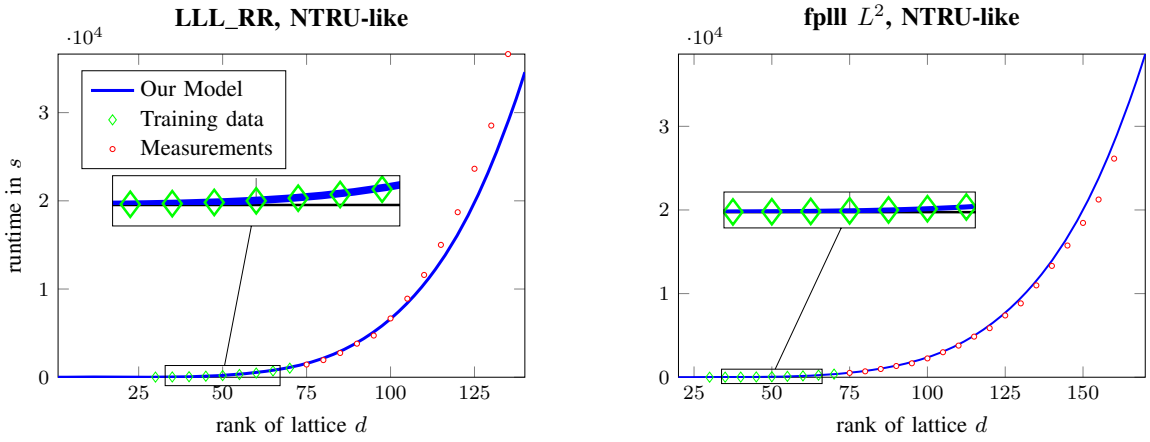

Figure 2: Extrapolation of runtime for NTRU-like matrices.

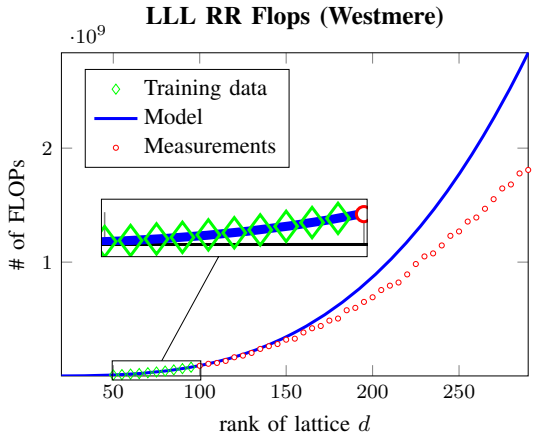

Figure 3: Extrapolation of the # of FLOPs for LLL_RR and the measurements for higher ranks on the Westmere system.

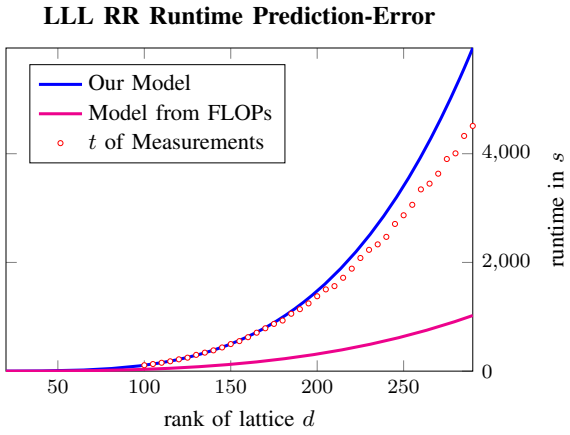

Figure 4: The real runtime and the runtime prediction resulting from the FLOPs.

The operations on the 150 bit types we used need to be mapped to standard 32 and 64 bit operations for the CPU floating point unit. Hence, the library which implements the arbitrary precision type as well as the configuration, performance and features of the compiler it is translated with also play an important role for the overall performance of LLL. The complexity models based on bit-operations suffer from a comparable problem since it cannot be predicted how they will be mapped to standard floating point operations and it is not clear upfront how a real computer would process them. Lastly, the clock rate, which is treated as a constant in the approach of Equation 12, is not constant in modern CPUs. For example, the Haswell E5-2698 employed varies its clock rate between 2.30 GHz and 3.6 GHz depending on the thermal conditions.

We see that modeling the runtime via FLOPs is inherently inaccurate for several reasons: First, the actual model that predicts the number of FLOPs may be inaccurate, like the model of [7] overestimates the actual number by a factor higher than $O(d^2)$. Second, assuming a linear dependence of the FLOPs and the runtime is misleading since various factors influence the time required to execute a fixed number of FP operations. This demonstrates that even a good prediction of the FLOPs, like in our case based on measurements, leads to unrealistic timing models. Furthermore, both types of inaccuracies are compounded for runtime prediction.

To summarize this section, Figure 5 exemplarily compares our model for the fplll library (same complexity class as NTL) and the estimate from [7] for random lattices with bit length d^2. Additionally, we show the model generated when all measurement points are used as training data. For the extrapolation, the predicted runtime differs for three orders of magnitude between the average model of [7] and our models but that our models nearly coincide for higher values.

Table II summarizes the exponents for the different models

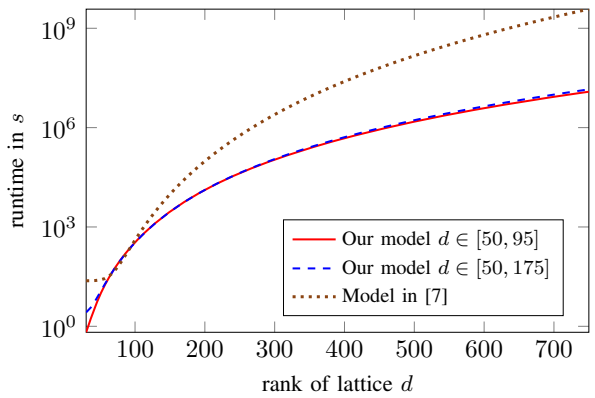

Figure 5: Comparison of the predicted runtime of our models for the fplll library and the average estimate from [7] on logarithmic scale for random lattices with bit length d^2.

of the form $o + c \cdot d^i \cdot \log^j d$ with the rank d as input parameter. The dominating part is the value of i.

Table II: Summary of the different performance models.

	SVP $10d$		SVP d^2		NTRU	
	i	j	i	j	i	j
NTL	3.05	3.49	4.38	2.50	4.40	2.50
fplll	3.07	3.50	4.75	2.49	5.25	0
Average [7]	6	0	8	0	8	0

VII. Summary and Outlook

We demonstrated a novel approach to generate performance models of algorithms with an unpredictable behavior with the use-case of the LLL algorithm. The quality of our models was verified and the different behavior for both implementations (NTL and fplll) resulting from the type of the input matrix employed was demonstrated. The models can be generated with low effort and in an automated fashion. Additionally, we compared both implementations for the same input matrix type and showed that for random lattices both libraries have the same behavior considering accuracy of the model and the derived complexity class while the absolute runtime differs because of the coefficients in the model functions. The case of NTRU-like matrices shows that for this type of matrices both LLL implementations behave differently and are a very good example that the knowledge of the complexity class is not sufficient to predict the actual runtime. For all matrix types, our models have a considerably lower complexity than delivered by the established average case estimate [7]. This means that problems may be solvable more than 1000 times faster than the average case estimate predicts.

We modeled the number of FLOPs for FP LLL and found that while we could predict FLOPs well with our model, again showing a lower complexity than previous estimates, the runtime could not be realistically predicted based on the clock rate alone. We generally doubt that this route is promising given the complexity of today's high-performance systems.

In the future, we will integrate the precision of the floating point data type as a second parameter to the performance models. We plan to apply the methods presented to other algorithms in the field of lattice-based cryptography (BKZ, enumeration [1]), and other general parallel algorithms where a prediction of the runtime is usefull, like resource allocation on HPC systems for simulation algorithms.

References

[1] 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.

[2] D. Stehlé, *Floating-Point LLL: Theoretical and practical aspects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–213.

[3] C. Ling, W. H. Mow, and N. Howgrave-Graham, "Reduced and fixed-complexity variants of the LLL algorithm for communications," *IEEE Transactions on Communications*, vol. 61, no. 3, pp. 1040–1050, March 2013.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[5] A. K. Lenstra, H. W. Lenstra, and L. Lovasz, "Factoring polynomials with rational coefficients," *MATH. ANN*, vol. 261, pp. 515–534, 1982.

[6] P. Q. Nguyen and D. Stehlé, "An LLL algorithm with quadratic complexity," *SIAM Journal on Computing*, vol. 39, no. 3, pp. 874–903, 2009.

[7] P. Q. Nguyen and D. Stehlé, *LLL on the Average*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 238–256.

[8] C. P. Schnorr, "A more efficient algorithm for lattice basis reduction," in *Automata, Languages and Programming*, L. Kott, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 359–369.

[9] A. Novocin, D. Stehlé, and G. Villard, "An LLL-reduction algorithm with quasi-linear time complexity," in *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '11. New York, NY, USA: ACM, 2011, pp. 403–412.

[10] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, November 2013, pp. 1–12.

[11] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? On the scalability of OpenMP implementations," in *Proc. of the 21st Euro-Par Conference, Vienna, Austria*, ser. Lecture Notes in Computer Science, vol. 9233. Springer, Aug. 2015, pp. 451–463.

[12] M. K. Ilyas, A. Calotoiu, and F. Wolf, *Off-Road Performance Modeling – How to Deal with Segmented Data*. Cham: Springer International Publishing, 2017, pp. 36–48.

[13] H. Cohen, *A Course in Computational Algebraic Number Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1993.

[14] P. Q. Nguyen and D. Stehlé, *Floating-Point LLL Revisited*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 215–233.

[15] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[16] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 42–42.

[17] D. Goldstein and A. Mayer, "On the equidistribution of Hecke points," *Forum Mathematicum*, no. 15, pp. 165–189, 2003.

[18] R. Dolbeau, "Theoretical peak FLOPS per instruction set on modern Intel CPUs," 2015. [Online]. Available: http://cs.iupui.edu/~fgsong/cs590HPC/how2decide_peak.pdf