CIS 400 Senior Project
Spring 2018 Final Report

1 Student Information

Group #18: Paul Lou.

Project title: ParaLLL: Parallelizing the LLL Algorithm

2 Advisor Information

Advisor: Dr. Nadia Heninger. We've have weekly lab meetings on Wednesday at 11AM with the rest of the lab present as well, though Dr. Heninger was away at conferences a few times and I skipped two weekly meetings (once in March, once in April) due to coursework. Dr. Heninger is also present regularly on other days for chats about the project since I'm usually in the lab at least 3 out of the 5 week days working on this project.

Mentor: Luke Valenta. Luke and I usually worked together on Fridays, sitting together to discuss relevant papers and write/edit code.

3 Summary

We aim to parallelize and implement the parallelized Lenstra-Lenstra-Lovász lattice basis reduction algorithm (LLL) for cluster-distributed cryptanalysis on Lattice-based cryptography schemes, reducing real-time computation time for any algorithms involving LLL-reduced basis.

4 Overview of Problem and Approach

A **Euclidean lattice** is a discrete subgroup of n-dimensional Euclidean space. Lattice-based cryptography is a subfield of cryptography in which lattice-based cryptosystems rely on the hardness of problems in Euclidean Lattices.

One such problem is the **shortest vector problem**: given a basis for the lattice, find the shortest non-zero vector of minimal Euclidean norm in the lattice. Solving the exact version of this problem has been shown to by NP-Hard.

We can solve the approximation version in time polynomial in the lattice dimension n using the **Lenstra-Lenstra-Lovász lattice basis reduction algorithm (LLL)**. LLL produces an "LLL-reduced" basis whose shortest basis is an $2^{O(n)}$-approximation on the shortest-vector. In practice, LLL beats the upper bound, facilitating practical cryptanalysis of non-lattice-based cryptography schemes such as the Merkel-Hellman knapsack cryptosystem and in cryptanalysis

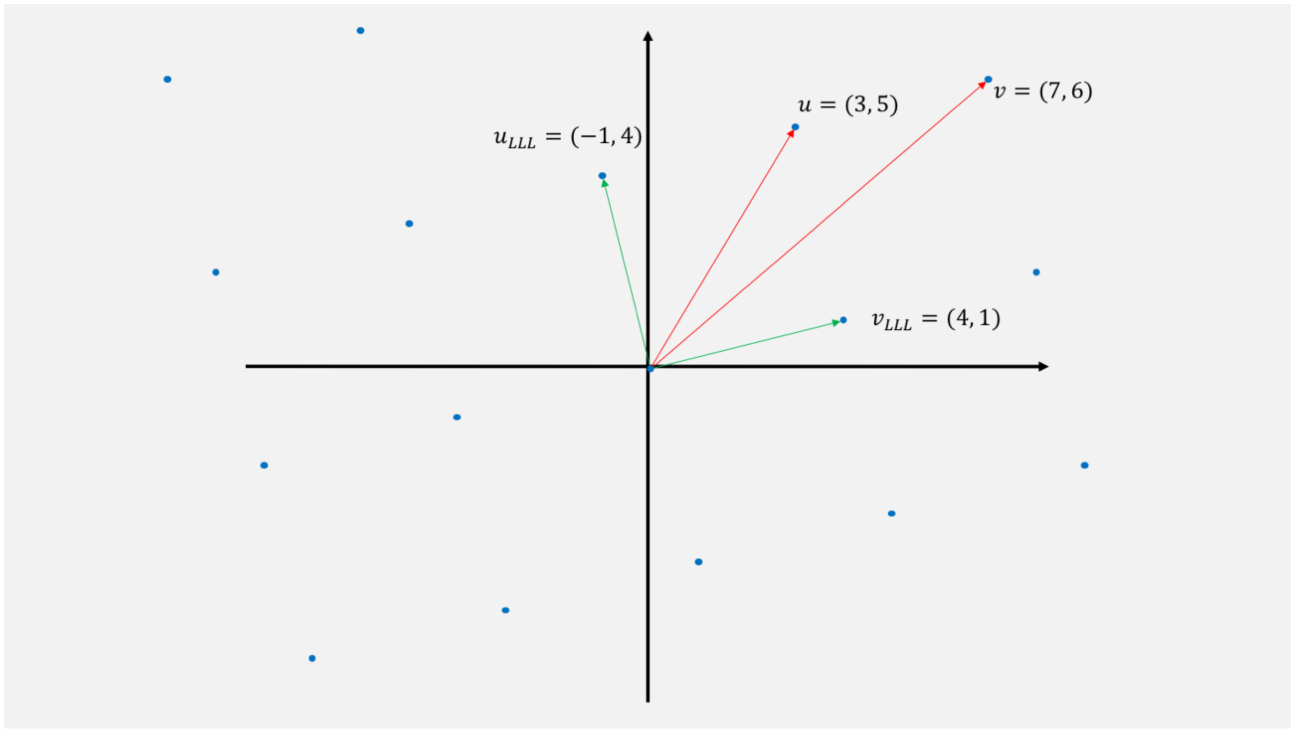


Figure 1: Two-dimensional visualization of LLL Reduction

on the RSA cryptosystem. Despite its prevalence, LLL is the runtime bottleneck in its applications, with a runtime of $O(n^5 \log^3 B)$ where $B = \max_i\{|b_i|\}$ [1].

We begin by understanding bottlenecks within the LLL algorithm and then we implement and experiment with variants of the LLL algorithm to understand how LLL can be effectively parallelized.

5 Implementation

Other than fast and parallel arithmetic operations, three primary approaches are of interest in understanding how to parallelize LLL: translations, input manipulation, and swaps. Of these three, the first two were tested the past semester and third one is implemented but testing is still underway.

We implemented in Python the classic LLL and a floating-point variant of the LLL algorithm. The original LLL algorithm worked with integers, but integer representation and operations become computationally expensive relative to floating-point operations. Still, Victor Shoup mentions in his number theory library NTL that the integer variant of LLL can result in faster runtimes compared to the floating-point variant on some types of inputs. In general, LLL's behavior on various types of lattices is not well understood. However, slight modifications to the the size-reduction step must be made to address numerical instability of the Gram-Schmidt procedure. We briefly describe the LLL algorithm and its components to motivate our experiments. Let $b_1, \ldots, b_n$ form a linearly independent basis for lattice Λ of dimension n and let $b_1^*, \ldots, b_n^*$ be their Gram-Schmidt orthogonalization. The Lovász Condition is as follows:

$$|b_{i+1}^* + \mu_{i+1,i} b_i^*|^2 \geq \delta |b_i^*|^2 \tag{1}$$

The Lovász Condition encapsulates the goal of producing basis vectors whose norms are within some function of δ from each other and preserving some degree of orthogonality as described the Gram-Schmidt coefficient μ_{i+1}. The LLL algorithm runs its main loop until all basis vectors satisfy the condition.

Algorithm 1 General LLL

```
1: Given: basis $b_1, \ldots, b_n$, reduction factor $\delta$
2: Size reduce $b_1, \ldots, b_n$ (alg. 2)
3: if any $b_j$ does not satisfy eqn. (1) then
4:     swap $b_j$ and $b_{j+1}$
5:     goto step 2
6: end if
7: return LLL-reduced $b_1, \ldots, b_n$
```

Algorithm 2 Gram-Schmidt Size Reduction

```
 1: Given: basis $b_1, \ldots, b_n$, reduction factor $\delta$
 2: Compute Gram-Schmidt coefficients $\mu_{i,j}$
 3: for $i = 2$ to $n$ do
 4:     for $j = i - 1$ downto 1 do
 5:         $b_i \longleftarrow b_i - \lceil \mu_{i,j} \rfloor b_j$
 6:         for $k = 1$ to $j$ do
 7:             $\mu_{i,k} \longleftarrow \mu_{i,k} - \lceil \mu_{i,j} \rfloor \mu_{j,k}$
 8:         end for
 9:     end for
10: end for
```

Input manipulation is an attempt to break the input basis vectors into blocks to perform a mergesort-esque LLL reduction. The mergeLLL algorithm runs the C++ fplll for each LLL subroutine. A discussion on different merge algorithms is saved for the evaluation.

Algorithm 3 MergeLLL

```
1: Given: basis $b_1, \ldots, b_n$, reduction factor $\delta$
2: $E \leftarrow LLL(\{b_{2k} : k \in \mathbb{Z}, 2k \in [1, n]\})$
3: $O \leftarrow LLL(\{b_{2k-1} : k \in \mathbb{Z}, 2k - 1 \in [1, n]\})$
4: $M \leftarrow merge(E, O)$
5: return $LLL(M)$
```

Aiming to parallelizing swaps leads to us implementing the following algorithm due to Villard [2]. Testing this implementation is not finished.

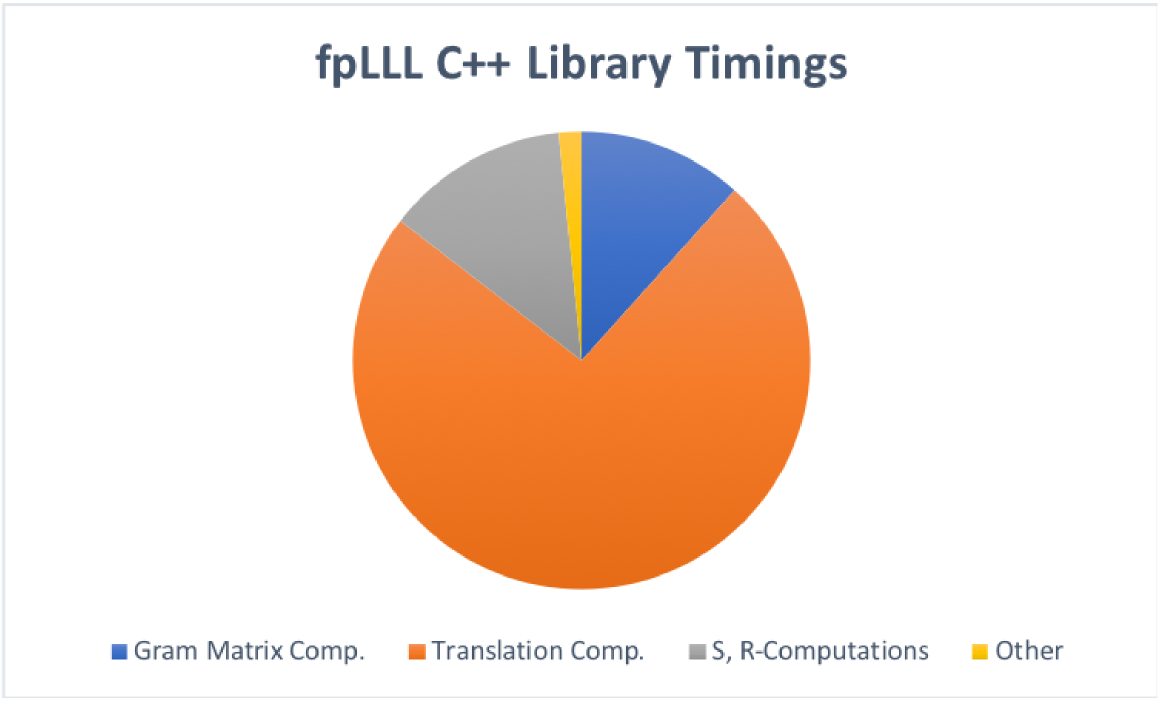
Algorithm 4 General even-odd LLL

```
1:  Given: basis $b_1, \ldots, b_n$, reduction factor $\delta$
2:  Size reduce $b_1, \ldots, b_n$ (alg. 1)
3:  while swaps remain do
4:      for even $j$ simultaneously do
5:          if $b_j$ does not satisfy eqn. (1) then
6:              swap $b_j$ and $b_{j+1}$
7:          end if
8:      end for
9:      Size reduce $b_1, \ldots, b_n$ (alg. 1)
10:     Repeat steps 3 through 7 for odd $j$
11: end while
12: return LLL-reduced $b_1, \ldots, b_n$
```

Note that all relevant variants of LLL algorithm follow the general structure involving size reduction and swaps defined by algorithm 1.

6 Evaluation

On each iteration of the LLL algorithm, an iteration being the execution of lines 2 through 4 of the General LLL algorithm, the most computationally expensive step is the size-reduction. Within size reduction (algorithm 2), line 5 is the translation step and the takes up the most time. We benchmarked the runtime of each component of the floating point variant of LLL and provide a visualization of the relative runtime of the translation step (line 5 in algorithm 2) in fplll, a state-of-the-art C++ implementation of the floating-point variant of the LLL algorithm.



Note that in all our following experiments, the inputs were tested on random lattices generated by producing lower triangular matrices were random inputs whose bit sizes were grown by adding randomly chosen rows to each other.

A cursory glance suggests a naive parallelization method on the translations in which each thread performs rounding on a Gram-Schmidt coefficient and one multiplication. The result of each threads computation is then joined by subtracting the results out of the basis at the current working index. However, this approach resulted in runtimes that were three to five times slower on our Python implementation of floating point LLL with multiprocessing compared to without multiprocessing. Such a result confirms the suspicion that translation computations happen too frequently and the overhead of managing threads at this deeply nested line of code is too expensive for any runtime speedups.

Ruling out naive translation parallelization, we turn to a simpler approach: input manipulation. The nature of the LLL algorithm is similar to that of a sorting algorithm with additional orthogonality constraints. In fact, the original integer version of LLL acts as a slow bubblesort. A natural thing to try is to apply a divide-and-conquer approach. We divide a given linearly independent basis $\{b_1, b_2, \ldots, b_n\}$ into two blocks, B_1, B_2 where we make the ad hoc decision that B_1 contains the vectors with odd indices and B_2 contains the vectors with even indices. We apply the LLL algorithm on B_1 and B_2 in parallel, resulting in two sets of LLL-reduced basis vectors $LLL(B_1)$ and $LLL(B_2)$. Two different merge steps were tested, the stack merge and the interleave merge. The stack merge is the naive merge algorithm, placing B_1 on top of B_2 to produce $\{B_1, B_2\}$. Since the LLL algorithm's output is sorted by basis norm, however, intuition suggests that an interleaving merge in which a vector is picked in an alternating fashion from B_1 and B_2 can perhaps better satisfy the Lovász conditions, removing a few iterations of the LLL algorithm.

Dim.	**Vanilla fpLLL**	**Stack**	**Interleave**
50	20.5	21.8	26.3
100	183.5	190.0	239.8
150	790.6	759.1	775.8

Table 1: Average wall time in seconds with Sage's LLL()

Reported timings are averaged across 5 iterations. Timings were volatile and input dependent due to uncontrolled orthogonality between even and odd indexed basis elements. We see slight speedups in higher dimensions on the average from the stack and interleave merge methods. Note, however, stacking and interleaving on individual trials beat the Vanilla fpLLL up to 61% on wall-clock time.

Finally, Villard's even-odd algorithm provides a theoretical framework for parallelizing swaps. We have implemented an alternate version of the integer LLL algorithm in C++ using the NTL library that performs parallelized swaps. Testing, however, is still underway.

In summary, we evaluate and analyze two different approaches to parallelizing the LLL algorithm through the translation step and input manipulation. We show that the translation step is ineffective due to threading overhead but we show that the divide-and-conquer approach provides speedups on the average on lattices of dimension 150 and the overhead of splitting and merging vectors decreases as the dimension increases, suggesting increased speedups as the dimension increases. Parallelizing swaps is trickier but more promising and tests are underway.

7 Lattice Cryptography in the Real World

Although the LLL algorithm provides a way to attack a variety of problems, lattice cryptography in general offers the construction of post-quantum security and a basis on which to construct useful cryptographic tools such as fully homomorphic encryption. According to the 2017 Official Annual Cybercrime Report sponsored by the Herjavec Group, cybercrime will cost the world $6 trillion annually by 2021 [3]. This cost and risk will only increase with the advent of adoption of the internet of things with not only wearables but entire public infrastructure systems. Much of the deployed cryptography relies on problems such as the discrete log problem or integer factorization which are unknown to be NP-Hard. However, Shor's algorithm is a polynomial-time quantum algorithm that would efficiently solve both problems. While it is most likely not the case that the advent of quantum computing would end practical security offered by such cryptography in an instant, quantum computing is pushing the cryptography community to explore "post-quantum cryptography", which is defined as cryptography that is usable today but still safe in the presence of a fully operational quantum computer. Lattice cryptography is one of many avenues through which post-quantum cryptoschemes can be constructed. Other avenues include multilinear maps and supersingular isogenies, but of these approaches, the usage of lattices is older, more established, and has more of a proven resilience to cryptanalytical attempts.

Fully homomorphic encryption (FHE) can be informally described in the following scenario. A user wants to compute some function on some data in the cloud but does not want the cloud server to know the plain data. FHE allows the user to send her encrypted data and the server to reply with the result of the function on the encrypted data where the decryption of the result is the result of the function on the non-encrypted data. Lattice theory enabled the construction of the first FHE scheme by Gentry in 2009. Yet, the existence of algorithms such as LLL has posed a risk to existing schemes. As such, FHE is not ready for industrial use as it fails to satisfy both usable security and efficient compute-times.

Parallelizing LLL, then, aims to find greater vulnerabilities in lattice based cryptoschemes and constructs. Having it as an open source library aids researchers in their cryptanalysis. Detecting such vulnerabilities protects both industry and the public against misplaced trust in existing and newly proposed schemes.

8 Individual Contributions

Most of the work reported above is performed by Paul. Luke modified the C++ fplll code to parallelize translations with OpenMP and noticed that just compiling the C++ fplll with OpenMP or pthreads caused a 15x overhead on our Intel machines. This overhead was confirmed by Dr. Daniel J. Bernstein in the Netherlands on an AMD machine. Luke also obtained Paul Kircher's parallelized lattice basis reduction algorithm and his partially written paper from Kircher's advisor. He ran some inputs on Kircher's algorithm and showed that Kircher's algorithm performs extremely fast compared to sequential fplll. On a dimension 500 lattice fplll finishes reducing the lattice in 46 hours while Kircher's finishes in 37 minutes. We are not quite sure what how the algorithm works, but what is clear from the paper is that it is a combination of a whole bunch of existing techniques.

Dr. Heninger provided the overall direction throughout this ongoing project.

References

[1] *The LLL Algorithm.* Springer, 2009.

[2] G. Villard. Parallel lattice basis reduction. *ISSAC*, July 1992.

[3] Morgan S. Cybercrime damages $6 trillion by 2021. *Cybersecurity Ventures*, 2017.

[4] P. Q. Nguyen and D. Stehlé. An lll algorithm with quadratic complexity. *SIAM Journal on Computing*, 2009.

[5] B. Werner and S. Wetzel. Parallel lattice basis reduction - the road to many-core. *High Performance Computing and Communications (HPCC)*, 2011.

[6] Albrecht et al. fplll. `https://github.com/fplll/fplll`, 2018.