Sieving in practice: The Generalized Sieve Kernel (G6K)

Elena Kirshanova

based on joint work with Martin R. Albrecht, Leo Ducas, Eamonn W. Postlethwaite, Gottfried Herold, Marc Stevens

The Simons Institute for the Theory of Computing
May 5, 2020
Part 0

Preliminaries
Given \(\{b_1, \ldots, b_n\}\) – a basis of \(L\), the Shortest Vector Problem (SVP) asks to find non-zero \(v\) of minimal length. We do not know \(||v|\|_{\text{shortest}}\) in general, but for any \(n\)-rank \(L\): 
\[||v|\|_{\text{shortest}}| \leq \sqrt{n \cdot \det(L)^{1/n}}\] (Minkowski's bound)
Short vectors in $\mathcal{L}$

Given $\{b_1, \ldots, b_n\}$ – a basis of $\mathcal{L}$, the Shortest Vector Problem (SVP) asks to find non-zero $v$ of minimal length.

We do not know $\|v_{\text{shortest}}\|$ in general, but for any $n$-rank $\mathcal{L}$:

$$\|v_{\text{shortest}}\| \leq \sqrt{n} \cdot \det(\mathcal{L})^{1/n} \quad \text{(Minkowski’s bound)}$$
Lattices of our interest

Goldstein-Mayer type of lattice with a basis given by columns:

\[ B = \begin{pmatrix} p & x_1 & \ldots & x_{n-1} \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}, \]

where \( p \) is a large prime and \( x_i \) are iid. uniform random from \( \{0, \ldots, p - 1\} \).

\[ \det(L(B)) = p \implies \| \mathbf{v}_{\text{shortest}} \| \leq \sqrt{np^{1/n}} \]

We’ll be fine with \( \mathbf{v} \) slightly longer than the shortest, e.g.,

\[ \| \mathbf{v} \| \approx 1.05 \cdot \sqrt{n} \det(L)^{1/n} \quad (1.05 - \text{Hermite-SVP}) \]
Projected lattice

\[ \cdot b_1, \ldots, b_n \text{ – basis of } \mathcal{L} \]
\[ \mathbf{b}_1, \ldots, \mathbf{b}_n - \text{basis of } \mathcal{L} \]
- $b_1, \ldots, b_n$ – basis of $\mathcal{L}$
- $b_i^\star$ is the projection of $b_i$ on $L_{1:i-1}^\perp$. (GSO)

- $b_1 = b_1^\star$
- $b_1, \ldots, b_n$ – basis of $\mathcal{L}$
- $b^*_i$ is the projection of $b_i$ on $\mathcal{L}_{1:i-1}^\perp$ (GSO)
- $b_1, b^*_2, \ldots, b^*_n$ - GSO basis of $\mathcal{L}$

Projected lattice
Projected lattice

- $\mathbf{b}_1, \ldots, \mathbf{b}_n$ – basis of $\mathcal{L}$
- $\mathbf{b}_i^*$ is the projection of $\mathbf{b}_i$ on $\mathcal{L}_{1:i-1}^\perp$ (GSO)
- $\mathbf{b}_1, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*$ – GSO basis of $\mathcal{L}$
- often it is convenient to represent lattice vectors in GSO basis
What is inside g6k

The General Sieve Kernel implements

1. Exact-SVP
   The output is compared against the Gaussian heuristic

2. 1.05-Hermite SVP
   Darmstadt SVP-Challenge

3. BKZ

4. LWE
   Darmstadt LWE-Challenge
What is inside g6k

The General Sieve Kernel implements

1. **Exact-SVP**
   The output is compared against the Gaussian heuristic

2. **1.05-Hermite SVP**
   Darmstadt SVP-Challenge

3. **BKZ**

4. **LWE**
   Darmstadt LWE-Challenge

Each of the above can use either of the following sieve:

- `gauss_sieve`
- `nv_sieve` (Nguyen-Vidick sieve)
- `bjg1` (single- or multi-threaded) (Becker-Gama-Joux bucket sieve)
- `triple_sieve` (single- or multi-threaded)
Part I

nv_sieve
Nguyen-Vidick sieve

All sieving algorithms start by **sampling** lots of lattice vectors into a list \( L \) and by **sorting** it.

- **Sampling** can be done by 1. sampling the last coordinates \( \text{wrt. GSO basis} \) and 2. lifting to the full lattice using Babai
Nguyen-Vidick sieve

All sieving algorithms start by **sampling** lots of lattice vectors into a list $L$ and by **sorting** it.

- **Sampling** can be done by 1. sampling the last coordinates wrt. GSO basis and 2. lifting to the full lattice using Babai
- **Sieving** searches for pairs $x, y \in L$ s.t. $\|x \pm y\|$ is small

$L$

$x$

$y$
Nguyen-Vidick sieve

All sieving algorithms start by sampling lots of lattice vectors into a list $L$ and by sorting it.

- **Sampling** can be done by 1. sampling the last coordinates wrt. GSO basis and 2. lifting to the full lattice using Babai
- **Sieving** searches for pairs $x, y \in L$ s.t. $\|x \pm y\|$ is small
- Once found replace the longest vector in $L$ with $x \pm y$

![Diagram of list $L$ with elements $x$, $y$, and $x \pm y$]
Nguyen-Vidick sieve

All sieving algorithms start by **sampling** lots of lattice vectors into a list $L$ and by **sorting** it.

- **Sampling** can be done by 1. sampling the last coordinates wrt. GSO basis and 2. lifting to the full lattice using Babai
- **Sieving** searches for pairs $x, y \in L$ s.t. $\|x \pm y\|$ is small
- Once found replace the longest vector in $L$ with $x \pm y$
- The next short $x' \pm y'$ replaces the current longest on $L$

![Diagram of list $L$ with sampled vectors $x$, $y$, $x' \pm y'$, and $x \pm y$]
All sieving algorithms start by sampling lots of lattice vectors into a list $L$ and by sorting it.

- **Sampling** can be done by 1. sampling the last coordinates wrt. GSO basis and 2. lifting to the full lattice using Babai

- **Sieving** searches for pairs $x, y \in L$ s.t. $\|x \pm y\|$ is small

- Once found replace the longest vector in $L$ with $x \pm y$

- The next short $x' \pm y'$ replaces the current longest on $L$

- Stop when “enough” short pairs are found

Runtime: $|L|^2$. For $|L| = \left(\frac{4}{3}\right)^{n/2}$, $T = 2^{0.415n}$
How to efficiently discard unpromising pairs, [Cha02, FBB+15, Duc18]

We spend most of the time testing if $\mathbf{x} \pm \mathbf{y}$ is short.
Need to compute the scalar product $|\langle \mathbf{x}, \mathbf{y} \rangle|$ fast
Most scalar products are useless, leading to no new vectors.
How to efficiently discard unpromising pairs, [Cha02, FBB+15, Duc18]

We spend most of the time testing if $x \pm y$ is short. Need to compute the scalar product $|\langle x, y \rangle|$ fast. Most scalar products are useless, leading to no new vectors.

Close vectors are likely to lie in the same half-space. Quick test with XOR + Popcount. We implement this test for all sieves.
How to efficiently discard unpromising pairs, [Cha02, FBB+15, Duc18]

We spend most of the time testing if $x \pm y$ is short. Need to compute the scalar product $|\langle x, y \rangle|$ fast. Most scalar products are useless, leading to no new vectors.

Close vectors are likely to lie in the same half-space. Quick test with XOR + Popcount. We implement this test for all sieves.
How to efficiently discard unpromising pairs, [Cha02, FBB+15, Duc18]

We spend most of the time testing if $\mathbf{x} \pm \mathbf{y}$ is short. Need to compute the scalar product $|\langle \mathbf{x}, \mathbf{y} \rangle|$ fast. Most scalar products are useless, leading to no new vectors.

Close vectors are likely to lie in the same half-space. Quick test with XOR + Popcount. We implement this test for all sieves.
Part II

bjg1
bjg1 = NV Sieve + Buckets

Bucket center $x \in L$ defines a region $B_x$. 
Bucket center \( \mathbf{x} \in L \) defines a region \( B_\mathbf{x} \)

Bucketing phase

\( \forall \mathbf{y} \in L : \)

If \( \langle \mathbf{x}, \mathbf{y} \rangle \geq \alpha : \)

If \( \| \mathbf{x} \pm \mathbf{y} \| \) small
  reduce \( \mathbf{y} \)

Else put \( \mathbf{y} \) into \( B_\mathbf{x} \)
Bucketing phase

\( \forall y \in L : \)

If \( \langle x, y \rangle \geq \alpha : \)

If \( \|x \pm y\| \) small

reduce \( y \)

Else put \( y \) into \( B_x \)

Sieve the bucket

\( \forall y \in B_x : \)

Find \( y' \in B_x \) s.t.

\( \|y \pm y'\| \) – small
Bucket center \( x \in L \) defines a region \( B_x \)

**Bucketing phase**

\( \forall y \in L : \)
- If \( \langle x, y \rangle \geq \alpha \):
  - If \( \|x \pm y\| \) small
    - reduce \( y \)
  - Else put \( y \) into \( B_x \)

**bjg1 strategy:**

choose \( x \) randomly from \( L \)
for \( 2^{0.142n} \) centres to find all pairs:

\[
T = 2^{0.349n}, \ M = 2^{0.2075n}
\]

**Sieve the bucket**

\( \forall y \in B_x : \)
- Find \( y' \in B_x \) s.t.
  - \( \|y \pm y'\| \) – small
Bucketing phase

\[ \forall y \in L : \]
- If \( \langle x, y \rangle \geq \alpha \):
  - If \( \|x \pm y\| \) small
    - reduce \( y \)
  - Else put \( y \) into \( B_x \)

BGDL strategy:

- choose \( x \) from a spherical code
  - \( T = 2^{0.292n}, M = 2^{0.2075n} \)
- decoding random spherical code introduces overheads

Sieve the bucket

\[ \forall y \in B_x : \]
- Find \( y' \in B_x \) s.t.
  - \( \|y \pm y'\| \) small
Parallelized bjg1

- Parallelization is done by having different threads work with different buckets.
- Reading the database of vectors is lock free.
- Insertions of new shorter vectors into the global list are delayed and are executed in batches.
- Sorting is complicated.
Part III

triple_sieve
**Triple sieve**

Motivation: reduce $2^{0.2075n}$ memory

3-Sieve searches for triples $x, y, z \in L$ s.t. $\|x \pm y \pm z\|$ is small

- Once found replace the longest vector in $L$ with $x \pm y \pm z$
- Stop when “enough” short pairs are found

Memory optimal regime: $M = 2^{0.1887n}, T = 2^{0.3588n}$

Generalises to $k$-Sieve but taking $k > 3$ seems impractical
Bucketing phase

∀ y ∈ L:
If ⟨x, y⟩ ≥ α:
  If ∥x ± y∥ small
    reduce y
  Else put y into B_x

Sieve the bucket

∀ pairs y, y', ∈ B_x:
  If ∥y ± y'|| - small
    perform the reduction
triple_sieve vs. bjg1

Bucketing phase

\( \forall y \in L : \)

If \( \langle x, y \rangle \geq \alpha' : \)

If \( \|x \pm y\| \) small

reduce \( y \)

Else put \( y \) into \( B_x \)

\cdot tuning the parameters allows to interpolate btw. 2-Sieve and 3-Sieve

\cdot Smaller list = \Rightarrow more 3-reductions

\cdot Larger list = \Rightarrow more 2-reductions
triple_sieve vs. bjg1

triple_sieve

Bucketing phase

\( \forall y \in L : \)

If \( \langle x, y \rangle \geq \alpha' \):

- If \( \|x \pm y\| \) small
  - reduce \( y \)
- Else put \( y \) into \( B_x \)

Sieve the bucket

\( \forall \) pairs \( y, y' \), \( \in B_x : \)

- If \( \|y \pm y'\| \) small
  - perform the reduction
- Else if \( \|x \pm y \pm y'\| \) small
  - perform the reduction

· tuning the parameters allows to interpolate btw. 2-Sieve and 3-Sieve

· Smaller list = more 3-reductions

· Larger list = more 2-reductions
triple_sieve vs. bjg1

**triple_sieve**

**Bucketing phase**

\[ \forall y \in L : \]

If \( \langle x, y \rangle \geq \alpha' \):

- If \( \|x \pm y\| \) small
  - reduce \( y \)
- Else put \( y \) into \( B_x \)

**Sieve the bucket**

\[ \forall \text{ pairs } y, y', \in B_x : \]

If \( \|y \pm y'\| \) - small

- perform the reduction

Else if \( \|x \pm y \pm y'\| \) - small

- perform the reduction

- tuning the parameters allows to interpolate btw. 2-Sieve and 3-Sieve
- Smaller list \( \implies \) more 3-reductions
- Larger list \( \implies \) more 2-reductions
Time-memory trade-offs

With the same $|L|$, `triple_sieve` finds more reductions than 2-Sieve. It allows to decrease $|L|$.

The right most point corresponds to the 2-Sieve memory regime.
The left most – to the 3-Sieve memory regime.
Parallelized *triple_sieve*

- Again parallelization is done by having different threads work with different buckets.
- Reading the database of vectors is lock free
- Insertions of new shorter vectors into the global list are delayed and are executed in batches
Part IV

The G6K framework
The G6K framework

G6K includes previous and introduces new improvements for sieving:

1. **Progressive sieving.** [Duc18,ML18]: iteratively sieve in projected sublattices of smaller dimension

2. **Dimensions for free,** [Duc18]: sieve in a projected sublattice, Babai-lift short vectors

3. **BKZ with sieving:** recycle short vectors from one projected lattice to another (do not start from scratch!)
The G6K framework

G6K includes previous and introduces new improvements for sieving:

1. Progressive sieving. [Duc18, ML18]: iteratively sieve in projected sublattices of smaller dimension

2. Dimensions for free, [Duc18]: sieve in a projected sublattice, Babai-lift short vectors

3. BKZ with sieving: recycle short vectors from one projected lattice to another (do not start from scratch!)

Sources of improvements:

- Sieve outputs not only one short vector but many short vectors
- Sieving tries to improve the “quality” of a basis rather than just finding a shortest vector
- “Quality” - length of Gram-Schmidt vectors
- $\mathbf{b}_1, \ldots, \mathbf{b}_n$ - basis of $\mathcal{L}$
- $\mathcal{L}_{1:j}$ is the lattice spanned by $\mathbf{b}_1, \ldots, \mathbf{b}_j$. 

$\mathbf{b}^*_{i:j} = \mathbf{b}_i \cap \mathcal{L}_{1:j}$ is the projection of $\mathbf{b}_i$ on $\mathcal{L}_{1:j}$. 

$\mathcal{L}_{i:j}$ is the orthogonal projection of $\mathcal{L}_{1:j}$ on $\mathcal{L}_{1:i-1}$. 

$\mathbf{b}_1 = \mathbf{b}^*_1$. 

$\mathbf{b}_2$ and $\mathbf{b}^*_2$. 

$\mathcal{L}_{1:1}$. 

Projected lattice
Projected lattice

- \( \mathbf{b}_1, \ldots, \mathbf{b}_n \) – basis of \( \mathcal{L} \)
- \( \mathcal{L}_{1:j} \) is the lattice spanned by \( \mathbf{b}_1, \ldots, \mathbf{b}_j \).
- \( \mathbf{b}^*_i \) is the projection of \( \mathbf{b}_i \) on \( \mathcal{L}_{1:i-1}^\perp \) (GSO)
- \( \mathcal{L}_{i:j} \) is the orthogonal projection of \( \mathcal{L}_{1:j} \) on \( \mathcal{L}_{1:i-1}^\perp \).
Non black-box sieving in G6K

- Sieve in a projected sublattice $\mathcal{L}_{i:j}$ of rank $j - i + 1$. The output a list $L$ of short vectors.
- Short vectors can be lifted from $\mathcal{L}_{i:j}$ to $\mathcal{L}_{i':j}$ for $i' < i$.
- Particularly short lifts are inserted into the current basis.
- Move from $\mathcal{L}_{i:j}$ to $\mathcal{L}_{i':j'}$ using the set of instructions:
  - **Lifting**: moves from $\mathcal{L}_{i:j}$ to $\mathcal{L}_{i-1:j}$
  - **Inclusion**: moves from $\mathcal{L}_{i:j}$ to $\mathcal{L}_{i:j+1}$
  - **Projection**: moves from $\mathcal{L}_{i:j}$ to $\mathcal{L}_{i+1:j}$
Non black-box sieving in G6K

With this set of instructions

- **Lifting**: moves from $L_{i:j}$ to $L_{i-1:j}$
- **Inclusion**: moves from $L_{i:j}$ to $L_{i:j+1}$
- **Projection**: moves from $L_{i:j}$ to $L_{i+1:j}$

G6K implements

1. **Progressive sieving**, [Duc18,ML18] iteratively sieve in $L_{i:n}$ for decreasing $i$’s.
Non black-box sieving in G6K

With this set of instructions

– **Lifting**: moves from $L_{i:j}$ to $L_{i-1:j}$
– **Inclusion**: moves from $L_{i:j}$ to $L_{i:j+1}$
– **Projection**: moves from $L_{i:j}$ to $L_{i+1:j}$

G6K implements

1. **Progressive sieving**, [Duc18,ML18]: iteratively sieve in $L_{i:n}$ for decreasing $i$’s.
2. **Dimensions for free**, [Duc18]: sieve in $L_{f:n}$ until enough short vectors are found, lift to $L_{1:n} = \mathcal{L}$.
   For $f = \mathcal{O}(n/\lg n)$, lifts include the shortest vector.
Non black-box sieving in G6K

With this set of instructions

- **Lifting**: moves from $L_{i:j}$ to $L_{i-1:j}$
- **Inclusion**: moves from $L_{i:j}$ to $L_{i:j+1}$
- **Projection**: moves from $L_{i:j}$ to $L_{i+1:j}$

G6K implements

1. **Progressive sieving**, [Duc18,ML18]: iteratively sieve in $L_{i:n}$ for decreasing $i$’s.

2. **Dimensions for free**, [Duc18]: sieve in $L_{f:n}$ until enough short vectors are found, lift to $L_{1:n} = L$.
   For $f = O(n/\lg n)$, lifts include the shortest vector.

3. **Pumping**: progressive sieve from $L_{n:n}$ to $L_{i:n}$, insert $n - i + 1$ short vectors.
Non black-box sieving in G6K

With this set of instructions

- **Lifting**: moves from $L_{i:j}$ to $L_{i-1:j}$
- **Inclusion**: moves from $L_{i:j}$ to $L_{i:j+1}$
- **Projection**: moves from $L_{i:j}$ to $L_{i+1:j}$

G6K implements

1. **Progressive sieving**, [Duc18,ML18] iteratively sieve in $L_{i:n}$ for decreasing $i$’s.

2. **Dimensions for free**, [Duc18]: sieve in $L_{f:n}$ until enough short vectors are found, lift to $L_{1:n} = L$.
   For $f = O(n/\lg n)$, lifts include the shortest vector.

3. **Pumping**: progressive sieve from $L_{n:n}$ to $L_{i:n}$, insert $n - i + 1$ short vectors.

4. **Workout**: execute Pump for decreasing $i$’s.
Non black-box sieving in G6K

With this set of instructions

- **Lifting**: moves from $L_{i:j}$ to $L_{i-1:j}$
- **Inclusion**: moves from $L_{i:j}$ to $L_{i:j+1}$
- **Projection**: moves from $L_{i:j}$ to $L_{i+1:j}$

G6K implements

1. **Progressive sieving**, [Duc18,ML18] iteratively sieve in $L_{i:n}$ for decreasing $i$'s.

2. **Dimensions for free**, [Duc18]: sieve in $L_{f:n}$ until enough short vectors are found, lift to $L_{1:n} = L$. For $f = \mathcal{O}(n/\lg n)$, lifts include the shortest vector.

3. **Pumping**: progressive sieve from $L_{n:n}$ to $L_{i:n}$, insert $n - i + 1$ short vectors.

4. **Workout**: execute Pump for decreasing $i$'s.

5. **BKZ**
BKZ (simplified)

\[ \mathcal{L}_{[\ell; r]} \text{ - orthogonal projection of } \mathcal{L}_{1:r} \text{ on } \mathcal{L}_{1:\ell-1} \]

**Input:** \( B = (b_i), \beta \)

for \( k = 2 \ldots n - 1 \) do

\( b \leftarrow \text{SVP}(\mathcal{L}_{[k : \min\{k+\beta-1, n\}]}) \)

end for

if \( b \) is “short enough” then

Insert \( b \) into \( B \)

Remove lin. dependencies

end if
BKZ with Sieving

\[ \mathcal{L}_{[\ell:r]} \] - orthogonal projection of \( \mathcal{L}_{1:r} \) on \( \mathcal{L}_{1:1}^\perp \)

**Input:** \( B = (b_i), \beta \)

**for** \( k = 2 \ldots n - 1 \) **do**

- Sieve(\( \mathcal{L}_{[k;k+1]} \))
- Sieve(\( \mathcal{L}_{[k;k+2]} \))
- \ldots
- Sieve(\( \mathcal{L}_{[k;k+\beta]} \))

**end for**

**Update** \( B \) with short \( b_i \)'s from \( \mathcal{L}_{[k;k+\beta]} \)
BKZ with Sieving

$L_{[l:r]}$ - orthogonal projection of $L_{1:r}$ on $L_{1:l-1}^\perp$

Input: $B = (b_i), \beta$

for $k = 2 \ldots n - 1$ do

Sieve($L_{[k;k+1]}$)
Sieve($L_{[k;k+2]}$)

...  
Sieve($L_{[k;k+\beta]}$)

end for

Update $B$ with short $b_i$'s from $L_{[k;k+\beta]}$
The last part

Experimental results
Experimental results (bgj1_1)

<table>
<thead>
<tr>
<th>SVP dim</th>
<th>Hermite factor</th>
<th>Sieve max dim</th>
<th>Wall time</th>
<th>Total CPU time</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>155</td>
<td>1.00803</td>
<td>127</td>
<td>14d 16h</td>
<td>1056d</td>
<td>246 GiB</td>
</tr>
<tr>
<td>153</td>
<td>1.02102</td>
<td>123</td>
<td>11d 15h</td>
<td>911d</td>
<td>139 GiB</td>
</tr>
<tr>
<td>151</td>
<td>1.04411</td>
<td>124</td>
<td>11d 19h</td>
<td>457.5d</td>
<td>160 GiB</td>
</tr>
<tr>
<td>149</td>
<td>0.98506</td>
<td>117</td>
<td>60h 7m</td>
<td>4.66kh</td>
<td>59 GiB</td>
</tr>
<tr>
<td>147</td>
<td>1.03863</td>
<td>118</td>
<td>123h 29m</td>
<td>4.79kh</td>
<td>67.0 GiB</td>
</tr>
<tr>
<td>145</td>
<td>1.04267</td>
<td>114</td>
<td>39h 3m</td>
<td>1496h</td>
<td>37.7 GiB</td>
</tr>
</tbody>
</table>

On various machines with a lot of RAM (256 or 512 GiB). The current record due to L. Ducas, M. Stevens, W. van Woerden: dim = 157
Implementation

The G6K is implemented as a C++ and Python library and is open-source.

https://github.com/fplll/g6k

The paper is available at

https://eprint.iacr.org/2019/089
Implementation

The G6K is implemented as a C++ and Python library and is open-source.

https://github.com/fplll/g6k

The paper is available at

https://eprint.iacr.org/2019/089

Thank you!

Q?
References

- [Cha02] M. Charikar. Similarity estimation techniques from rounding algorithms.
- [Duc18] L. Ducas, Shortest vector from lattice sieving: A few dimensions for free.
- [NV08] P. Nguyen, T. Vidick. Sieve algorithms for the shortest vector problem are practical.