Enhancing the scalability and memory usage of HashSieve on multi-core CPUs

Artur Mariano
Institute for Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
artur.mariano@sc.tu-darmstadt.de

Christian Bischof
Institute for Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
christian.bischof@sc.tu-darmstadt.de

***Abstract*—The Shortest Vector Problem (SVP) is a key problem in lattice-based cryptography and cryptanalysis. While the cryptography community has accumulated a vast knowledge of SVP-solvers from a theoretical standpoint, the practical performance of these algorithms is commonly not well understood. This gap in knowledge poses many challenges to cryptographers, who are oftentimes confronted with algorithms that perform worse in practice then expected from theory. This is a problem because the asymptotic complexity of the best algorithms plays a key role in the construction of cryptosystems, but only practically appealing, validated algorithms are accounted for in this process. Thus, if one cannot extract the full potential of theoretically strong algorithms in practice, efficient algorithms might be ruled out and wrong assumptions are made when constructing cryptosystems.**

In this paper, we take a step forward to fill this gap, by providing a computational analysis of HashSieve, the most practical sieving SVP-solver to date, and showing how its performance can be enhanced in practice. To this end, we revisit the parallel generation of random numbers, memory allocation and memory access patterns. Employing scalable random sampling, object memory pools, scalable memory allocators and aggressive memory prefetching, we were able to improve the best current implementation of HashSieve by factors of 3x and 4x, depending on the lattice dimension, and set new records for the HashSieve algorithm, thereby shrinking the gap between its theoretical complexity and its performance in practice.

I. Introduction

Three decades ago, the cryptography community engaged in an intensive search for cryptosystems that would be resistant against attacks with quantum computers, due to the vulnerability of classical cryptosystems, such as RSA, against these attacks. Today, lattice-based cryptography stands out as one of the most prominent post-quantum types of cryptography, for several reasons. First of all, lattice-based cryptosystems are efficient and very simple to implement. In addition, it is known that lattice-based cryptosystems enjoy worst-case hardness, a powerful property for cryptosystems. Roughly speaking, this means that breaking the cryptosystem is at least as hard as solving several lattice problems in the worst case. Third, lattices can be used to implement fully-homomorphic encryption, a promising technique for cryptosystems.

Lattices are discrete subgroups of the n-dimensional Euclidean space $\mathbb{R}^n$, with a strong periodicity property. A lattice $\mathcal{L}$ generated by a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$,...,$\mathbf{b}_m$ in $\mathbb{R}^n$, is denoted by: $\mathcal{L}(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^{m} \mathbf{u}_i \mathbf{b}_i, \mathbf{u} \in \mathbb{Z}^m\}$, where m is the rank and n is the dimension of the lattice.

Lattice-based cryptosystems become vulnerable when certain mathematical problems (in this case lattice problems) are solved. To estimate the actual computational complexity of these problems in practice is of prime importance, because the parameters of cryptosystems are chosen based on this complexity. Overestimating the computational complexity of these problems forces one to use overly strong parameters, which might render the scheme impractical. On the other hand, underestimating their computational complexity might lead to vulnerable cryptosystems. This is the core reason why highly optimized, parallel solvers of the underlying lattice problems of lattice-based cryptosystems deserve study.

The central problem in this context is the Shortest Vector Problem (SVP), which consists in finding the non-zero vector $\mathbf{v}$ of a given lattice $\mathcal{L}$, whose Euclidean norm $\|\mathbf{v}\|$ is the smallest among the norms of all non-zero vectors in the lattice $\mathcal{L}$ and is denoted by $\lambda_1(\mathcal{L})$. It is well known that the SVP is NP-hard, so no polynomial time exact algorithms are expected to be be found. We refer to an algorithm that solves this problem as an *SVP-solver*. Although there are several types of SVP-solvers, enumeration and sieving are the most relevant classes of these solvers in practice. In particular, a combination of enumeration-based algorithms with extreme pruning and efficient lattice reduction algorithms seems to be the *de facto* approach to solve the SVP in high dimensions.

In fact, enumeration algorithms have been studied for several decades, with good results, which somewhat stunt the headway made on sieving algorithms. GaussSieve has been the most practical sieving algorithm, and many implementations, both on shared- and distributed-memory, were published (e.g. [3], [4]). However, even the best implementations of GaussSieve are not competitive with highly pruned enumeration on random lattices.

Very recently, there was a small turnaround in this matter, with the proposal of HashSieve [7]. HashSieve is an algorithm that simplifies the reduction process of GaussSieve, thus becoming considerably faster. A parallel implementation attaining good scalability on 16-core shared-memory systems was proposed shortly after, proving that sieving might be way more practical than previously believed [2]. Although there are records of sieving implementations used to break high dimensional lattices on the online SVP-challenge www.latticechallenge.org/svp-challenge/, this is, to our knowledge, the best implementation of sieving algorithms today.

Although enumeration and sieving algorithms have been considerably studied (although at different scales), little was done to understand their performance on modern computer architectures. In this paper, we present a computational analysis of HashSieve and we show how to improve the scalability and memory usage of HashSieve, with optimizations that can be extended to other sieving algorithms. To this end, we employ scalable random sampling mechanisms, object memory pools, scalable memory allocators and aggressive memory prefetching. As a result, we attained a considerable overall speedup in comparison to the implementation proposed in [2] and we set new records for the HashSieve algorithm.

Contributions. This paper shows, for the first time, an analysis of an SVP-solver, HashSieve, from a computational perspective, presenting its arithmetic intensity and memory access pattern. By applying techniques known to HPC, we show that the performance of HashSieve can be four times higher than previously reported, which unveils the full potential of the algorithm. These findings are meaningful because this imputes a greater value to sieving algorithms while attacks to lattice-based cryptosystems, thus forcing to use their complexity while selecting parameters for lattice-based cryptosystems.

Structure. The rest of this paper is organized as follows. Section II presents the HashSieve algorithm. Section III introduces the test platforms. Section IV briefly recaps the implementation presented in [2]. The following sections present our enhancements to HashSieve. In particular, Sections V, VI and VII present the improvement on the scalability, memory usage and tractability over the baseline implementation. Section VIII concludes the paper and presents future lines of research.

II. HashSieve

The core idea behind sieving algorithms is the reduction of vectors against one another [5]. Reducing one vector **v** against (a multiple of) another vector **w** means that **w** is either subtracted or added to **v**, so that **v** becomes shorter. The number of vectors necessary for convergence is not known upfront for the majority of sieving algorithms. The reduction of vectors is usually done in a brute-force manner: each vector is tested against all the other vectors, by means of an inner product, to determine whether the reduction is successful. In general, the closer these vectors are in Euclidean space (which is given by the inner product), the higher the probability for reduction.

HashSieve simplifies this process by filtering out a large number of these vectors. This is done with a popular method from nearest neighbor search, known as *locality-sensitive hashing*. This method organizes vectors in hash tables according to the distance spanned between them. For example, the vectors that are close in space to a vector **v** are stored in the same hash bucket of **v**, for every hash table. As each bucket is essentially some part of the lattice, the algorithm uses many hash tables to capture vectors that lie in the borders of those spaces. With this setup, the algorithm samples many vectors, as in every sieving algorithm, reducing them against the vectors with the same hash value, for every hash table. For a thorough description of HashSieve, we refer the reader to [7].

Algorithm 1: The HashSieve algorithm

```
1  Input: (Reduced) basis $\mathbf{B}$, collision threshold $c$;
2  Initialize stack $S \leftarrow \{\}$, collisions $cl \leftarrow 0$
3  Initialize $\mathtt{T}$ empty hash tables $\mathtt{H}_1, \dots, \mathtt{H}_T$

4  while $cl < c$ do
5      Pop vector $\mathbf{v}$ from $S$ or sample it if $|S| = 0$
6      while $\exists \boldsymbol{w} \in H_1[h_1(v)], ..., H_T[h_T(v)] : ||\boldsymbol{v} \pm \boldsymbol{w}|| < ||\boldsymbol{v}||$ do
7          for each Hash table $H_i, ..., H_T$ do
8              Obtain the set of candidates $\mathtt{C} = \mathtt{H}_i[h_i(\mathbf{v})]$
9              for each $\boldsymbol{w} \in C$ do
10                 Reduce $\mathbf{v}$ against $\mathbf{w}$ and Reduce $\mathbf{w}$ against $\mathbf{v}$
11                 if $\boldsymbol{w}$ has changed then
12                     Remove $\mathbf{w}$ from all $\mathtt{T}$ hash tables $\mathtt{H}_i$
13                     if $\mathbf{w} == 0$ then cl++
14                     else Add $\mathbf{w}$ to the stack $\mathtt{S}$

15     if $\mathbf{v} == 0$ then cl++
16     else Add $\mathbf{v}$ to all $\mathtt{T}$ hash tables $\mathtt{H}_i$
```

The pseudo-code of the HashSieve algorithm is given in Algorithm 1. After the initialization, the algorithm repeats the following four-step process: (i) sample a random lattice vector **v** (or get one from the stack S); (ii) find nearby candidate vectors **w** in the hash tables to reduce **v** with; (iii) use vector **v** to reduce other vectors **w** in the hash tables (and for each reduced **w**, move it onto the stack); and (iv) add **v** to the stack or hash tables. This process aims to build a large set of short, pairwise reduced vectors until two of them are $\lambda_1(\mathcal{L})$ apart. After that, the size of the set does not increase anymore, and collisions, which happen when vectors are reduced to the zero-vector, are generated instead. The algorithm terminates when a given number of collisions is reached.

The crucial difference between HashSieve and other sieving algorithms is how steps (ii) and (iii) are executed. Instead of traversing a list of vectors in linear time, the algorithm uses $\mathtt{T}$ independent hash tables $\mathtt{H}_1, \dots, \mathtt{H}_\mathtt{T}$ to look for nearby vectors. Given a target vector **v**, the algorithm performs these hash table look-ups by first computing the hash value $h_i(\mathbf{v})$ (where h_i is a locality-sensitive hash function, efficiently evaluated in $O(n^2)$), and then accessing the vectors in the hash table $\mathtt{H}_i$ that have the same hash value as **v**, i.e. all $\mathbf{w} : h_i(\mathbf{w}) = h_i(\mathbf{v})$, for all the hash tables $\mathtt{H}_i ... \mathtt{H}_\mathtt{T}$. These locality-sensitive hash functions enjoy a particular property: vectors mapped to the same hash bucket are more likely to be nearby than other vectors. This empowers HashSieve to intelligently filter vectors that are likely to be successfully used to reduced samples.

Increasing $\mathtt{K}$ and $\mathtt{T}$ renders the hash functions more selective. However, that comes at the cost of increasing the space usage because the algorithm holds more hash tables, with more buckets (each hash table has $2^\mathtt{K}$ buckets), and every vector needs to be stored in every hash table. The computational load is also increased, because one needs to compute the $\mathtt{T}$ hash values of each target vector **v** and perform $\mathtt{T}$ hash table look-ups. Thus, at some point, increasing the number of hash tables $\mathtt{T}$ does not improve the execution time any longer. In [7], it was shown that the choices $\mathtt{K} = \lfloor 0.2209n \rceil$ and $\mathtt{T} = \lfloor 2^{0.1290n} \rceil$ are sound. We will refer to these as optimal parameters, although they are not necessarily optimal in practice.

	16-core machine	60-core machine
#Sockets	2	4
CPU manufacturer	Intel	Intel
Model number	E5-2670	E7-4890 v2
Launch date	Q1'12	Q1'14
Micro-architecture	Sandy Bridge	Ivy Bridge
Frequency	2600 MHz	2800 MHz
Cores per chip	8	15
SMT	Hyper-threading	Turned off
L1 Cache	32 kB iC+dC	32 kB iC+dC
L2 Cache	256 kB	256 kB
L3 Cache	20 MB shared	37.5 MB shared
System memory	128 GBs	1 TB

TABLE I: Specifications of the test platforms.

III. Test platforms

Our analyses were carried out with several random lattices, generated with Goldstein-Mayer bases, in multiple dimensions, available on the SVP-challenge[1] website. All lattices were generated with seed 0. Table I provides the specifications of the two test platforms, with 16 and 60 cores, respectively (SMT stands for Simultaneous multi-threading and iC/dC for instruction/data cache). The 16-core machine runs Ubuntu 11.10, whereas the 60-core machine runs SUSE Linux ES.

The code was compiled with Intel's `icpc 13.1.3` on the 16-core machine and `icpc 15.0.2` on the 60-core machine, with the `-O2` optimization flag, since it was slightly better than `-O3`. The elapsed time of lattice reduction is not included in the results. The norm of the output vector of each and every run (sequential and parallel) was always the same.

The experiments were conducted with lattices in dimension 70 (seed 0) and onwards, since for reasonably strong basis reductions (aka reasonably high BKZ-β), a large part of the workload entailed by lower dimensions fits in the L3 cache of both machines, rendering the quantification of our memory optimizations less meaningful. We used the optimal HashSieve parameters for every experiment, except when said otherwise (e.g. Section VII). We reduced the lattice bases with BKZ from NTL[2]. The optimizations we report in this paper build upon the most efficient implementation of HashSieve to date, the implementation proposed in [2], which we refer to as the *baseline implementation* from here on. For the tests involving hardware counters, we used PAPI[3].

IV. Parallel HashSieve Implementation

In this section, we briefly review the baseline implementation published in [2], written in C. Essentially, the application spawns a team of threads, each of which samples a vector **v**, and tries to reduce **v** against **w** and vice versa, where **w** is one of the candidate vectors of **v**. The set of candidate vectors of a given vector **v** comprises the vectors that have the same hash value (i.e. are stored in the same hash bucket) of **v**, for each and every hash table in the system. If **v** is reduced against any of the candidate vectors, the thread repeats the whole process again (i.e. goes through all the hash tables, looking for candidates to reduce **v** against, most likely on different buckets). If **w** is reduced, it is removed from all the hash tables and moved onto the stack (which is private per thread).

Whenever a sample has gone through all the hash tables without being reduced, the thread inserts **v** in every hash table and samples another vector. The implementation employs a light-weight probable lock-free mechanism to handle concurrent accesses to the tables: whenever a thread accesses a hash bucket, it atomically sets a variable to 1, turning it to 0 when the visited bucket is no longer needed. The same system is used for regular vectors in the system, because as every hash table has one pointer to every vector in the system, the same vector might be accessed (and modified) from different hash buckets, by different threads (cf. Figure 1 in [2]). If a thread encounters one vector under use, the vector is simply disregarded, and the thread proceeds to the next iteration.

Arithmetic intensity. The kernels of the algorithm come down to (1) the *dot product* $\langle\mathbf{v},\mathbf{w}\rangle$, (2) the addition of one vector to another and (3) the calculation of the hash value of each vector (cf. [7]). The number of operations and accessed bytes in these kernels is summarized in Table II, where the arithmetic intensity (number of operations per byte fetched from memory) is calculated for a lattice in dimension 80. Note that in this process we ignored the arithmetic intensity of the loops. Also, we assume that there is only one deference of the array of coordinates as this is usually optimized by compilers, which create a *const* auxiliary pointer that points to the array.

Setting aside the other procedures in the algorithm, and assuming n = 80, the maximum arithmetic intensity of the analysed kernels is $\approx 1/6$, which is rather low. Furthermore, the arithmetic intensity decreases with the lattice dimension. The actual arithmetic intensity is way below this mark: these kernels aside, the algorithm essentially fetches high volumes of data from memory, so that these kernels can be executed. In particular, the algorithm (1) loads hash buckets, (2) adds and (3) removes vectors to/from hash buckets. These procedures are difficult to bound in terms of memory loads and stores, but they will only contribute to lower the overall arithmetic intensity. More than that, they contribute to very high cache miss ratios, which we report in the next subsection.

Calculating the ultimate contribution of (1), (2) and (3) to the overall arithmetic intensity is very hard, because the number of times they are executed varies considerably from lattice to lattice (even for the same dimension). Although we could use hardware counters to capture the memory traffic the algorithm ultimately incurs, there is no way to capture integer computations, since they are clouded by control-flow operations. Nonetheless, the point of this section is to prove that the algorithm is memory bound and that most of the optimization opportunity lies on the memory side, to which we dedicate Section VI. Additional evidence that the implementation is

Kernel	Dot product	Add	Hash Val
Operations	$2n$	$n + 3$	$3n/2 + 4$
Load/stored bytes	$12n + 16$	$6n + 36$	$10n + 8$
Arithmetic intensity	$\approx 1/6$	$\approx 1/6$	$\approx 1/7$

TABLE II: CPU operations and bytes fetched from memory for the three kernels of HashSieve, for n=80.

[1]www.latticechallenge.org/svp-challenge/
[2]www.shoup.net/ntl/
[3]http://icl.cs.utk.edu/papi/

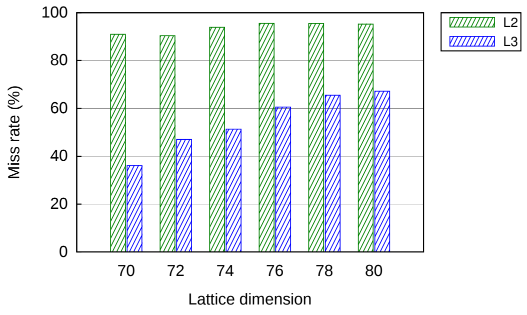

Fig. 1: L2/L3 miss ratios for the HashSieve implementation in [2], running with 32 threads on the 16-core machine. $\beta = 34$.

memory bound is the increase of performance when using SMT (cf. Figure 2), which does only benefit the performance of memory-bound applications.

Memory access irregularity. Irregular memory access patterns are well known for impairing performance. The memory access pattern of HashSieve is substantially irregular. To verify this claim, we measured the number of L2 and L3 cache misses in the baseline HashSieve implementation, for dimensions 70-80 in steps of 2, since irregular applications have very high miss ratios. These are shown in Figure 1. For smaller dimensions, BKZ finds the solution and HashSieve does not run for enough time to render this analysis useful.

The high cache misses ratios are due to the different hash tables that are consecutively visited, in order to reduce samples. There are $2^{\texttt{K}} \times \texttt{T}$ buckets that one thread can access, and the number of vectors looked at, per bucket, can be as low as one. Therefore, the likelihood of accessing the same hash bucket in a short time-frame is rather small. The miss ratios grow with the dimension because the higher the dimension, the larger the number of buckets per hash table $2^{\texttt{K}}$ and/or the number of hash tables $\texttt{T}$, which increases the likelihood of a cache miss. For a sufficiently high dimension (88 and onwards), we expect that both the L2 and L3 cache miss ratios are close to 100%.

In this section, we show that HashSieve is memory-bound, and one should optimize memory usage to improve performance. Section VI shows a few techniques to achieve that end.

V. Improving Scalability

Klein's algorithm (cf. [6]) is used to sample vectors in sieving algorithms. The first works on parallel sieving algorithms reported substantially faster convergence when increasing parameter d in Klein's algorithm up to some point. This is because Klein's algorithm samples shorter vectors on such a setup, which accelerates the convergence of sieving algorithms. However, it has also been reported that the scalability of some of these (shared-memory) implementations became hampered when increasing d. In particular, the optimal value of d when running these implementations in sequential (expectedly optimal in the parallel versions), rendered the implementations less scalable, and therefore less efficient [4], [2]. The degree to which increased values of d undermined scalability was higher for the implementation of HashSieve reported in [2] than for the implementation of GaussSieve reported in [4].

The fundamental cause of this behavior is the simultaneous use of the *rand*() function (*drand48*(), in these cases) by several threads. When one of these functions is used simultaneously by several threads, the actual behavior is undefined and depends on the actual implementation of these functions. One of the possibilities is the slow down of the code, since threads synchronize with one another to keep track of each others state. This was the behavior of the program reported on both papers. The connection with an increased parameter d in Klein's algorithm is straightforward: higher values of d render the sampler computationally heavier, as more iterations have to be performed for certain criteria to be verified. This results in more calls to *drand48*(), which aggravates the problem. This is more noticeable in HashSieve because the reduction process is faster, and so the likelihood for two threads to call the sampler (and thus the *drand48*() function) at the same time is higher.

To solve this problem, one can use random functions with private states (e.g. *drand48_r*()). This way, each thread has a private state of the randomization sequence and no synchronization with the other threads is necessary. We modified the implementation presented in [2] to account for this. The application creates one private state for each thread. The states are not aggregated into an array to avoid cache thrashing. It should be noted that using more than one random number generator, with different seeds, in parallel, does not necessarily mean that the generated numbers follow the original distribution (or even lead to random numbers in general). However, for this particular application, it suffices that the samplers output different vectors, which in fact happened otherwise too many collisions would be generated before the solution was gotten.

Figure 2(a) shows the scalability of the HashSieve implementation described in [2] with Klein's parameter d equals to 20 and 70. The performance of HashSieve with $d = 70$ is better for a single thread, but as the scalability of HashSieve is low when $d = 70$, almost flat-lining for more than 4 threads, a lower value for d delivers better results. With $d = 20$, the application scales linearly, and outperforms $d = 70$ for 8 or more HashSieve. Figure 2(b) shows the scalability of the implementation with a private random state per thread, for both d equals to 20 and 70. The performance of the implementation with $d = 20$ is identical to the original version, but the scalability for $d = 70$ became linear as well, outperforming $d = 20$ for any number of threads. The performance for one thread is different because different seeds for *drand48*() are used, which does not interfere with the scalability.

We also conducted experiments to verify the scalability of the improved implementation of HashSieve (with private states per thread) for higher core counts. Figure 2(c) shows its scalability on the 60-core machine. An analysis of how and why the performance of our implementation varies on both machines is out of the scope of this paper. These experiments aim to prove the high scalability of the improved implementation on higher core counts. The implementation scales linearly up to 60 threads (watch over the interruption in the x axis), if each thread is fed with a private status for the *drand48_r*() function, despite of what Klein's parameter d is used. With this implementation, we are able to run a lattice in dimension 80 (taken from the SVP-challenge, generated with seed 0) in 371 seconds with the 16-core machine (includes SMT) and in 189 seconds with the 60-core machine (without SMT).

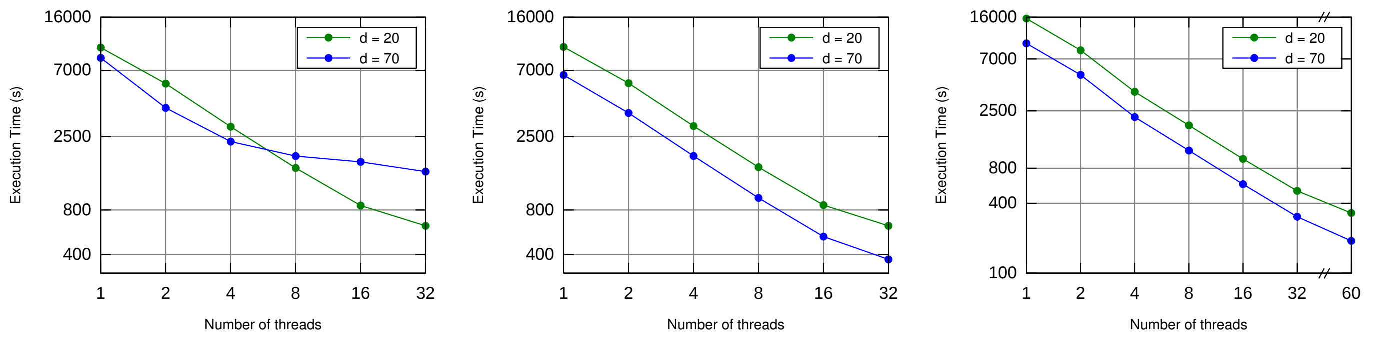

Fig. 2: Scalability of the implementation in [2] for Klein's parameter d equals to 20 and 70, for (a) original version on the 16-core machine, (b) private states per thread on the 16-core machine and (c) private states per thread on the 60-core machine.

This optimization represents already a speedup of ≈2.3x over the baseline implementation. From here on, every reported experiment will be built into this optimized version. In the next section, we show a series of optimizations, each of which builds onto the previous one, with incremental speedups.

VI. Optimization of Memory Usage

As mentioned before, memory usage is a critical aspect of HashSieve. In this context, there are three different aspects that can be identified. First, the number and size of the hash tables in the algorithm grows exponentially and linearly, respectively, with the lattice dimension. Although this leads to very high memory consumptions (e.g. dimension 100 would require 2.6 TB if the optimal parameters were to be used), this is an inherent characteristic of the algorithm and little can be done to mitigate this. However, there are two other aspects that one can identify and improve in the *baseline implementation*.

The first aspect is a particularly demanding memory allocation pattern. The baseline implementation allocates memory every time a new vector is to be sampled and whenever a bucket pool needs to be extended. Thus, a considerable part of HashSieve's runtime is spent on requesting these allocations to the operating system, which is typically a concurrent method if the default memory allocator *malloc* is used. The second aspect is that HashSieve is a memory bound algorithm, as we showed in Section IV, and most of its execution time is actually spent on fetching data from the memory hierarchy rather than processing it. In this section we show how one can tackle these problems and optimize HashSieve in practice.

A. Object Pools

The baseline implementation implements hash buckets with pools (of pointers to vectors), an attempt to mitigate the cost of adding vectors to the hash tables. These pools are re-sized whenever needed, with *realloc*. For samples, no pools are used: when vectors are sampled, they are stored with a *malloc* call. This is inefficient for a few reasons. First, this might lead to memory fragmentation because there is no assurance that these vectors will be stored contiguously in memory. Second, this incurs additional overhead because each of these *malloc* calls is an operating system call and they are invoked in parallel by several threads, which requires a locking mechanism to control concurrency. Third, this has reduced locality of reference.

We improved this with another pool of vectors, private per thread, this time used to store sample vectors. Essentially, we allocate a big array of vectors, in the beginning of the application, along with its size and its latest used position:

```
Vector *pool = malloc(POOL_SIZE*sizeof(Vector));
int pool_size = POOL_SIZE, latest_pos = 0;
```

Whenever a vector of the pool is used, the `pool_size` variable is decremented and the `latest_pos` variable is incremented. When a vector is used, the size of the pool is checked, and the pool is resized when it is empty.

The goodness of such a vector pool is two-fold. First, it minimizes the number of *malloc* calls. Second, it improves spacial locality, since vectors are consecutively stored in memory.

Table III shows the execution time of the baseline implementation, with and without a pool of samples. The achieved speedup is as much as 10.9%. It is impractical to extend these experiments to higher dimensions. The speedup grows with the lattice dimension, except for dimension 86. It is hard to speculate if the speedup will continue to grow for higher dimensions, because the benefit depends both on the number and the timing of each *malloc* call, which varies between executions. The gains are thus larger for more overlaps between these calls.

We also implemented an actual memory pool, i.e. a mechanism that allocates a very large space in memory and distributes chunks on that space upon request. The advantage of a memory pool is that only a single *malloc* call is done and retrieving chunks entails nothing but incrementing pointers. However, we did not implement a *realloc* function in our memory pool, which becomes a whole task on its own. Due to this reason, our performance gains were marginal, since when a *realloc* is to be made (to extend an hash bucket), each thread asks the pool for more memory and copies the elements from

Dimension	80	82	84	86
Without pool	371	777	1471	2738
With pool	355	730	1326	2530
Speedup	4,5%	6,4%	10,9%	8,2%

TABLE III: Execution time with and without pools of vectors for samples, on the 16-core machine (32 threads). BKZ-β=34.

Dimension	80	82	84	86
With pool	355	730	1326	2530
With pool + tcmalloc	348	699	1261	2528
Speedup	2,0%	4,4%	5,2%	<1%

TABLE IV: Execution time with and without `tcmalloc`, on the 16-core machine (running with 32 threads). BKZ-β=34.

the previous memory space to the freshly allocated memory space, thus incurring a substantial overhead.

B. *Generic memory allocators*

An alternative to memory pools are generic memory allocators. Although these mechanisms are considerably different in terms of aim and design, both usually accelerate memory allocation. Generic memory allocators are usually more scalable and more efficient than the default *malloc* system call, especially on multi-threaded applications. `Hoard`[4], and `tcmalloc`[5] are among the most popular generic memory allocators. Some, as `tcmalloc`, are drop-in replacements of *malloc* and other system calls, while others provide an API.

Our HashSieve implementation allocates memory in parallel in three different stages: (i) during the initialization, where all the hash tables and buckets are allocated, (ii) to create the various object pools (as discussed in subsection VI-A), private to each thread, and (iii) to extend the memory reserved to every bucket (with *realloc*). The latest is the most problematic among them, because even for dimension 80, millions of *realloc*s are in fact called. The number of *realloc*s is much bigger than the number of used vectors, which means that one vector is moved around in many hash tables throughout the application.

We chose `tcmalloc` for our experiments, due to two reasons. First, `tcmalloc` is a drop-in replacement to memory system calls, thus not requiring much effort to be integrated into the code. Second, it was the only allocator available on both benchmarking machines. As shown in Table IV, the use of `tcmalloc` is beneficial and increases with the lattice dimension. As also happened with the object pool, `tcmalloc` is less effective for dimension 86. We believe that this is a coincidence. We also ruled out the possibility of a correlation with the number of used (and allocated) vectors, since it spikes precisely from dimension 84 to dimension 86 (grows 57%), if compared to the growth from dimension 82 to dimension 84 (only 23%). If any correlation held, `tcmalloc` would deliver the most noticeable performance boost on dimension 84.

The actual speedup boost depends not only on the number of allocations that are performed, but also on the number of overlapping mallocs that are avoided, which varies from execution to execution. It should be noted that the algorithm performs differently for different lattices, even in the same dimension. To have reliable statistical data, we would have to run multiple lattices in the same dimension. This, however, is out of the scope of this paper, since the main purpose of these experiments is to identify if it is worthwhile to invest time in writing a custom memory allocator for this algorithm (and perhaps algorithms with the same computational behavior, as other sieving algorithms for the SVP).

[4]http://www.hoard.org/

[5]http://goog-perftools.sourceforge.net/doc/tcmalloc.html

Dimension	80	82	84	86
With pool + tcmalloc	348	699	1261	2528
With pool + tcmalloc + prefetching	312	644	1176	2330
Speedup	11,5%	8,5%	7,2%	8,5%

TABLE V: Execution time with and without data prefetching, on the 16-core machine (running with 32 threads). BKZ-β=34.

C. *Prefetching*

The last of the optimizations we applied was software-based prefetching, with hand-inserted prefetch directives. Although it is usually claimed that achieving speedups with such a scheme is rather hard for irregular applications [8], [1], we did use prefetching successfully in HashSieve.

There are many opportunities on HashSieve to prefetch data, which are not captured by the compiler because they depend on runtime values of subsequent iterations. A representative example is the removal of one vector from all the hash tables in the system, which is shown in the code below.

```
for(int t = 0; t < T; t++){
   hash_value = hash_function(w, t);
   bkt_remove(&HashTables[t][hash_value], w);
}
```

As the compiler does not know the subsequent value of hash_value, it cannot prefetch the data where HashTables of iteration t+1 resides at. As programmers, we can not only make it more explicit to the compiler, as we can also prefetch the data. This is achieved by replacing the previous loop with:

```
int hash_values[T];
for(int t = 0; t < T; t++){
    hash_values[t] = hash_function(w, t);
}
for(int t = 0; t < T-1; t++){
   //Prefetch HashTables[t+1][hash_values[t+1]!
   bkt_remove(&HashTables[t][hash_values[t]], w);
}
bkt_remove(&HashTables[t][hash_values[T]], w);
```

This calculates all the indexes upfront and prefetches the data needed in iteration i+1 in iteration i. As such, when iteration i+1 is executed, the data will already be in cache (or the latency will be small, since data is requested before). This can be (and is) replicated throughout the code, for insertion and removal of vectors from hash buckets, and other minor operations. The speedup is summarized in Table V.

D. *Brief wrap up and discussion*

In the previous subsections, we showed how the performance of HashSieve can be enhanced in practice with a number of memory optimizations. In particular, we conclude that: (1) the allocation of object pools improved the code as

Dimension	90	92	94	96	98	100	...	107
Probing?	No			Yes (1-level)			...	Yes (1-level)
BKZ-β	34				40		...	36
Klein's d	70						...	70
Sample Pool (k)	100	100	100	150	150	250	...	850
K	20	20	21	21	22	22	...	23
T	3126	3738	4470	465	533	637	...	1046
Optimal K/T?	Yes			No			...	No
Target norm?	No						...	No
Used vectors (k)	≈2425	≈2998	≈4501	≈5565	≈7050	≈10054	...	≈26658
Solution	2419	2440	2526	2522	2541	2571	...	2626
Time (h)	0.86	1.72	3.74	6.52	10.03	18.19	...	119.31
Memory (GB)	≈310	≈380	≈872	≈95	≈113	≈256	...	≈912

TABLE VI: Experiments with the optimized implementation of HashSieve on the 60-core machine, with 60 threads, for lattices in dimensions [90,107]. Statistics pertaining to the number of vectors for convergence, execution time of the application and the memory ultimately spent, as functions of the used sample pool (in thousands of vectors) and T and K parameters.

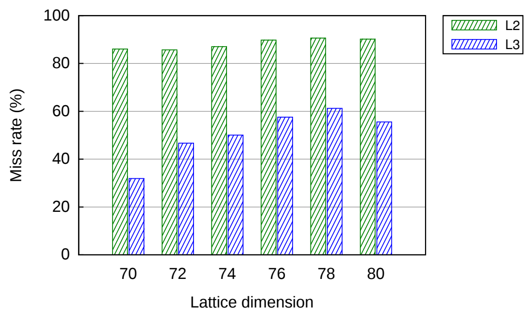

Fig. 3: L2/L3 miss ratios for the improved HashSieve implementation, on the 16-core machine (32 threads). BKZ-$\beta = 34$.

much as ≈11%, (2) tcmalloc, a generic memory allocator, provides even further speedup for the majority of the lattice dimensions we tested and (3) intensive prefetching does also boost performance, by pre-calculating the indexes of the following elements to be accessed and fetching them beforehand. With these optimizations, we are now able to run a lattice in dimension 80 in 151 and 312 seconds, respectively for the 16- and 60-core machines. As Figure 3 show, the both L2 and L3 miss ratios decreased with our optimizations.

The experiments conducted did not aim to select the best mechanisms for the optimizations we investigated (e.g. find the best generic memory allocator for our implementation), but rather identify generic lines of improvement in the algorithm (and its class). Also, we did not quantify memory consumption savings with the optimizations we implemented, since it would require a much more extensive set of benchmarks and analysis.

VII. Improving Tractability

In this section, we report benchmarks carried out on the 60-core machine, for lattices in higher dimensions (90-107). We used the most efficient HashSieve implementation, i.e., the version which includes the optimizations of Sections V and VI, running with 60 threads. The aim of this section is to redefine the tractability of HashSieve in high dimensions, in light of our optimizations. Therefore, we do not quantify the gains of the previously reported optimizations for these experiments.

For these benchmarks, we implemented one level probing, which mitigates the high levels of memory usage in HashSieve. For a complete explanation of probing, we refer the reader to the original paper of HashSieve [7]. Put simply, probing consists in visiting more buckets for each hash table, at the same time the number of hash tables is reduced. As a result, the computation in the algorithm increases, but memory usage decreases (and by a larger factor). In particular, the number of hash tables is reduced by a factor of $\texttt{K}/2+1$. The key idea is that, if one cannot use the optimal parameters K and T due to a physical RAM limitation, probing can be used to reduce the number of used Hash tables, thus allowing one to use higher values for K and T. In theory, the execution time increase due to the use of fewer Hash tables is outweighed by the use of higher values for K and T. In practice, this has never been validated to this day, a task we accomplished in this paper.

Table VI shows the performance of our implementation on the 60-core machine, for dimensions 90-107. We used different parameters for BKZ-β, the initial size of the sample pool described in Section VI-A, and Klein's parameter d was set to 70. The choice of BKZ-β was guided by intuition, since it would be impractical to find its best values. As for K, T, we used the optimal parameters until dimension 94, since the application spent less than 1 TB of RAM and no probing had to be used. For higher dimensions, we resorted to one-level probing. Dimension 96 is actually a good example to illustrate the efficacy of probing: the optimal parameters K=21, T=5345 would require a little bit over 1 TB of RAM, which is not possible in the used machine. Therefore, without probing, suboptimal parameters would have to be used. With K=20 and T=4087, the implementation took 7.10 hours, requiring ≈474GBs of RAM. With probing, we were able to use higher parameters, which rendered the implementation faster (6.52h), but more importantly, spending 5 times less memory.

The table shows the final number of used vectors (in thousands), execution time of the application (in hours) and the ultimately required memory (in GBs). One relevant caveat on these experiments is that one cannot infer functions for the growth of the execution time, number of used vectors and used memory, by simply taking all the benchmarks into account, as no probing and optimal parameters K and T were used for dimensions 90-94, but not for larger dimensions.

Dimension	90	92	94	96
Baseline imp. [2]	3.43	7.35	11.07	17.38
Optimized imp.	0.86	1.72	3.74	6.52
Speedup	3.99x	4.27x	2.96x	2.67x

TABLE VII: Execution time of the baseline and optimized HashSieve implementations, in hours, on the 60-core machine.

Although we do not report the running time of BKZ, a few caveats should be noted. In our experiments, we used the BKZ implementation from NTL, which is not parallel. While the lattice in dimension 96 was reduced in 20 minutes, dimensions 98 and 100, with $\beta = 40$, took approximately 12 hours (it is known that the running time of BKZ increases exponentially with β). However, from dimension 102 on, the library requires the use of BKZ_XD instead of BKZ_FP, which considerably increases the running time for BKZ. For instance, reducing dimensions 107 with $\beta = 36$ took around 2 days. For $\beta = 40$, which we believe to deliver a considerably better output, the process returned a segmentation fault after 6 days.

The current, optimized implementation is considerably faster than that published in [2]. In particular, for the same benchmark setup (machine, running threads, compiler, etc), we obtained a speedup of about 4x for dimension 90, and about 3x for dimensions 94 and 96, the highest dimension solved in [2]. A big contributor to these speedups is the improvement of the scalability of the implementation, shown in Section V. Table VII summarizes the final execution times of both implementations and provides the final speedup.

VIII. Conclusions

Although the cryptography community has acquired a vast knowledge of algorithms for the SVP, their practical performance is yet not well understood. This is critical because wrong assumptions about practical algorithms might be made while constructing a cryptosystem. In this paper, we took a step forward to fill this gap, by analyzing and optimizing HashSieve, the best sieving algorithm for the SVP in practice.

Results. We showed that HashSieve is a memory-bound algorithm, for which end we computed its arithmetic intensity and provided data pertaining to its memory pattern behavior. In particular, we conclude that its arithmetic intensity is bounded to $1/6$ (although certainly way below that mark in practice), and the L2 and L3 cache miss ratios are very high ($>$95% for L2 and $>$60% for L3, in dimension 80) and increase with the lattice dimension. We speculate that both ratios are close to 100% from dimension 88 onwards. Due to these figures, we conclude that the biggest optimization opportunity on HashSieve lies on the memory side.

Therefore, we conducted a series of memory optimizations on the best implementation to date, proposed in [2], at the same time we enhanced its scalability. The memory optimizations included object pools, generic memory allocators, and intensive prefetching. Implementing private states for randomization rendered the code highly scalable, for any Klein's parameter d. Overall, we obtained a speedup of as high as 4x in comparison to [2], for the same benchmark setup.

Meaning of results. To put these results in perspective, the highest dimension solvable by our implementation in less than 24h is now 100, instead of 96. Also, the baseline implementation would require approximately 147 days to solve dimension 120 on the 16-core machine, whereas the current implementation would require less than 50 days. It is important to note that since SVP-solvers are usually ran for several days and even months, even 10% gains are significant. For instance, solving dimension 107, which took about 5 days, would run in about 15 days without our improvements.

This means that, by applying relatively known HPC techniques to sieving algorithms, we proved them much more practical than previously believed. Thus, the crossing point between HashSieve and other SVP-solvers may occur for much lower lattice dimensions than previously expected. As a result, depending on the used lattice dimensions, the asymptotic complexity of our HashSieve implementation should be considered, instead of other, e.g. enumeration-based, algorithms.

Generalization of results. Our optimizations can be generalized to other sieving algorithms. In particular, the application of an object pool on GaussSieve (e.g. the implementations in [3], [4]) is straightforward and should deliver similar speedups. All our optimizations would be useful for the implementation described in [4]. For instance, we could prefetch consecutive elements in the list (which are not consecutively stored in memory, and are therefore not brought to cache in the same memory transfer) and use a scalable memory allocator. It is still relevant to consider improvements to GaussSieve because it can leverage special lattice structures, such as ideal lattices, whereas it is unclear if HashSieve can be adapted in a similar way. Finally, as HashSieve is expected to be improved via different hash functions, our improvements are important since we could simply integrate a better hash function in our implementation and benefit from all optimizations altogether.

Future work. We plan to implement a parallel version of BKZ 2.0 to speed up lattice reduction, which became a bottleneck for further experiments (e.g. we could not reduce the basis of the 107-dimensional lattice with an adequate β).

Acknowledgments. This work has been co-funded by the DFG as part of project P1 within the CRC 1119 CROSSING.

References

[1] John Mellor crummey et al. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001.

[2] Artur Mariano et al. Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In *44th International Conference on Parallel Processing*, Beijing, China, September 1-4, 2015.

[3] Joppe Bos et al. Sieving for shortest vectors in ideal lattices: a practical perspective. Cryptology ePrint Archive, Report 2014/880, 2014.

[4] Mariano et al. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. SBAC-PAD'14, 2014.

[5] Miklós Ajtai et al. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.

[6] Philip Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, SODA '00, pages 937–941, Philadelphia, PA, USA, 2000.

[7] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. CRYPTO 2015, To appear.

[8] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual ISCA*, pages 188–200, 1995.