A new generic algorithm for hard knapsacks (preprint)

Nick Howgrave-Graham[1] and Antoine Joux[2]

[1] 35 Park St, Arlington, MA 02474
nickhg@gmail.com
[2] DGA and Université de Versailles Saint-Quentin-en-Yvelines
UVSQ PRISM, 45 avenue des États-Unis, F-78035, Versailles CEDEX, France
antoine.joux@m4x.org

Abstract. In this paper, we study the complexity of solving hard knapsack problems, especially knapsacks with a density close to 1 where lattice based low density attacks are not an option. For such knapsacks, the current state-of-the-art is a 28-year old algorithm by Shamir and Schroeppel which is based on birthday paradox techniques and yields a running time of $\tilde{O}(2^{n/2})$ for knapsacks of n elements and uses $\tilde{O}(2^{n/4})$. We propose here several algorithms which improve on this bound, finally lowering the running time down to $\tilde{O}(2^{0.311\,n})$, while keeping the memory requirements reasonably low at $\tilde{O}(2^{0.256\,n})$. We also demonstrate the practicability of our algorithms with an implementation.

1 Introduction

The knapsack problem is a famous NP-hard problem which has often been used in the construction of cryptosystems. An instance of this problem consists of a list of n positive integers $(a_1, a_2, \cdots, a_n)$ together with another positive integer S. Given an instance, there exist two form of knapsack problems. The first form is the decision knapsack problem, where we need to decide whether S can be written as:

$$S = \sum_{i=1}^{n} \epsilon_i a_i,$$

with values of ϵ_i in $\{0, 1\}$. The second form is the computational knapsack problem where we need to recover a solution ϵ if at least one exists.

The decision knapsack problem is NP-complete (see [5]). It is also well-known that given access to an oracle that solves the decision problem, it easy to solve the computational problem with n oracle calls. Indeed, assuming that the original knapsack admits a solution, we can easily obtain the value of ϵ_n by asking to the oracle whether the subknapsack $(a_1, a_2, \cdots, a_{n-1})$ can sum to S. If so, there exists a solution with $\epsilon_n = 0$, otherwise, a solution necessarily has $\epsilon_n = 1$. Repeating this idea, we obtain the bits of ϵ one at a time.

Knapsack problems were introduced in cryptography by Merkle and Hellman [12] in 1978. The basic idea behind the Merkle-Hellman public key cryptosystem is to hide an easy knapsack instance into a hard looking one. The scheme was broken by Shamir [16] using lattice reduction. After that, many other knapsack based cryptosystems were also broken using lattice reduction. In particular, the low-density attacks introduced by Lagarias and Odlyzko [9] and improved by Coster et al. [3] are a tool of choice for breaking many knapsack based cryptosystems. The density of a knapsack is defined as: $d = n/\max_i a_i$. Asymptotically, random knapsacks with density higher than 1 are not interesting for encryption, because admissible values S correspond to many possibilities for ϵ. For this reason, knapsacks with density lower than 1 are usually considered. The Lagarias-Odlyzko low-density attack can solve random knapsack problem with density $d < 0.64$ given access to an oracle that solves the shortest vector problem (SVP) in lattices. Of course, since Ajtai showed in [1] that the SVP is NP-hard for randomized reduction, such an oracle is not available. However, in practice, low-density attacks have been shown to work very well when the SVP oracle is replaced by existing lattice reduction algorithm such as LLL[3] [10] or the BKZ algorithm of Schnorr [13]. The attack of [3] improves the low density condition to $d < 0.94$.

However, for knapsacks with density close to 1, there is no effective lattice based solution to the knapsack problem. In this case, the state-of-the-art algorithm is due to Shamir and Schroeppel [14] and runs in time $O(n \cdot 2^{n/2})$ using $O(n \cdot 2^{n/4})$ bits of memory. This algorithm has the same running time as a basic birthday based algorithm on the knapsack problem but much lower memory requirement. To simplify the notation of the complexities in the sequel, we extensively use the soft-Oh notation. Namely, $\tilde{O}(g(n))$ is used as a shorthand for $O(g(n) \cdot \log(g(n))^i)$, for any fixed value of i. With this notation, the algorithm of Shamir and Schroeppel runs in time $\tilde{O}(2^{n/2})$ using $\tilde{O}(2^{n/4})$ bits of memory.

In this paper, we introduce new algorithms that improve upon the algorithm of Shamir and Schroeppel to solve knapsack problems. The paper is organized as follows: in Section 2 we recall the algorithm of Shamir and Schroeppel, in Section 3 we present the new algorithms and in Section 4 we describe practical implementations on a knapsack with $n = 96$. Section 3 is divided into 4 subsections, in 3.1 we describe the basic idea that underlies our algorithm, in 3.2 we present a simplified version, in 3.3 we present an improved version and in 3.4 we describe our most complete algorithm. Finally, in Section 5 we present several extensions and some possible applications of our new algorithms.

2 Shamir and Schroeppel algorithm

The algorithm of Shamir and Schroeppel was introduced in [14]. It allows to solve a generic integer knapsack problem on n-elements in time $\tilde{O}(2^{n/2})$ using a memory of size $\tilde{O}(2^{n/4})$. To understand this algorithm, it is useful to first recall the basic birthday algorithm that can be applied on such a knapsack. For simplicity of exposition, we assume that n is a multiple of 4. Let $a_1, \ldots, a_n$ denote the elements in the knapsack and S be the target sum. We first construct the set $\mathcal{S}^{(1)}$ containing all possible sums of the first $n/2$ elements (from a_1 to $a_{n/2}$) and $\mathcal{S}^{(2)}$ be the set obtained by subtracting from the target S any of the possible sums of the last $n/2$ elements. It is clear that any collision between $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$ can be written as:

$$\sum_{i=1}^{n/2} \epsilon_i \, a_i = S - \sum_{i=n/2+1}^{n} \epsilon_i \, a_i,$$

[3] LLL stands for Lenstra-Lenstra-Lovász and BKZ for blockwise Korkine-Zolotarev

where all the ϵs are 0 or 1. Such an equality of course yields $S = \sum_{i=1}^{n} \epsilon_i\, a_i$, a solution of the knapsack problem.

Conversely, any solution corresponds to a collision of this form. With the basic algorithm, in order to find all collisions, we can, for example, sort both sets $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$ and read them simultaneously in increasing order.

The basic remark behind the Shamir and Schroeppel algorithm is that, in order to enumerate $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$ by increasing order, it is not necessary to store both sets. Instead, they propose a method that allows to enumerate the set using a much smaller amount of memory. Clearly, there is a nice symmetry between $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$, thus it suffices to show how $\mathcal{S}^{(1)}$ can be enumerated. Shamir and Schroeppel start by constructing two smaller sets. The first such set $\mathcal{S}_L^{(1)}$ contains all sums obtained by combining the leftmost elements a_1 to $a_n/4$. The second set $\mathcal{S}_R^{(1)}$ contains all sums of $a_{n/4+1}$ to $a_{n/2}$. Clearly, any element from $\mathcal{S}^{(1)}$ can be decomposed as a sum of an element from $\mathcal{S}_L^{(1)}$ and an element from $\mathcal{S}_R^{(1)}$. With these decompositions, some partial information about the respective order of elements is known *a priori*. Indeed, when σ_L is in $\mathcal{S}_L^{(1)}$, σ_R and σ'_R are in $\mathcal{S}_R^{(1)}$, then $\sigma_L + \sigma_R < \sigma_L + \sigma'_R$, if and only if, $\sigma_R < \sigma'_R$.

To use this partial information, Shamir and Schroeppel first sort $\mathcal{S}_L^{(1)}$ and $\mathcal{S}_R^{(1)}$. Then they initialize a sorted list of sums by adding to each element of $\mathcal{S}_L^{(1)}$ the smallest element in $\mathcal{S}_R^{(1)}$. Once this is done, their enumeration process works as follows: at each step, they take the smallest element in the list of sums and output it. Then, they update the sum and put back in the sorted list of sums at its new correct place. To update the sum, it suffices to keep the element from $\mathcal{S}_L^{(1)}$ unchanged and to replace the element of $\mathcal{S}_R^{(1)}$ by its immediate successor.

The main difficulty with this algorithm is that, in order to obtain the overall smallest current sum at each step of the algorithm, one needs to keep the list of sums sorted throughout the algorithm. Of course, it would be possible to sort after each insertion, but that would yield a very inefficient algorithm. Thankfully, by using sophisticated data structure such as balanced trees also called AVL trees, after their inventors Adelson-Velsky and Landis, it is possible to store a sorted list of elements in a way that fits our needs. Indeed, these structures allow to maintain a sorted list where insertions and deletions can be performed in time logarithmic in the size of the list.

Since $\mathcal{S}_L^{(1)}$ and $\mathcal{S}_R^{(1)}$ have size $2^{n/4}$, so does the sorted list of sums. As a consequence, Shamir and Schroeppel algorithm runs in time $\tilde{O}(2^{n/2})$ like the basic birthday algorithm but only use a memory of size $\tilde{O}(2^{n/4})$.

2.1 Application to unbalanced knapsacks

With ordinary knapsacks, where no *a priori* information is known about the number of elements that appear in the decomposition of S, the various sets $\mathcal{S}^{(1)}$, $\mathcal{S}^{(2)}$, $\mathcal{S}_L^{(1)}$ and $\mathcal{S}_R^{(1)}$ are easy to define and everything works out well. With unbalanced knapsacks, we know that S is the sum of exactly ℓ elements and we say that the knapsack has weight ℓ. If $\ell = n/2$, the easiest theoretical solution is to ignore the information, because asymptotically it does not significantly help to solve the problem. However, even in this case, the additional information can significatively speed-up implementation. For the moment, we stick with an asymptotic analysis and we assume, without loss of generality[4] that $\ell < n/2$. For simplicity of exposition, we also assume that ℓ is a multiple of 4.

In this context, it seems natural to define $\mathcal{S}^{(1)}$ as the sum of all subsets of cardinality $\ell/2$ of the first $n/2$ knapsack elements a_1 to $a_{n/2}$. However, if the correct solution is not perfectly balanced by the two halves (or between the four quarters), it will be missed. For example, a solution involving $\ell/2+1$ element for the first half and $\ell/2-1$ from the second half cannot be detected. To avoid this problem, Shallue proposed in [15] to repeat the algorithm several times, randomizing the order of the a_is each time. The number of orders which lead to a correct decomposition of the solution into four quarter of size $\ell/4$ is:

$$\binom{\ell}{\ell/4 \; \ell/4 \; \ell/4 \; \ell/4} \cdot \binom{n-\ell}{(n-\ell)/4 \; (n-\ell)/4 \; (n-\ell)/4 \; (n-\ell)/4}.$$

And the total number of possible splits into four quarter is $\binom{n}{n/4 \; n/4 \; n/4 \; n/4}$. Using Stirling's formula, we find that the fraction of correct orders is polynomial in n and ℓ. As a consequence, we only need to repeat the algorithm polynomially many times.

Assuming that ℓ is written as $\alpha\, n$ we obtain an algorithm with time complexity $\tilde{O}(\binom{n/2}{\ell/2})$ and memory complexity $\tilde{O}(\binom{n/4}{\ell/4})$. Expressed in terms of α and n, the time complexity is $\tilde{O}(\mathcal{T}_\alpha^n)$ and the memory complexity $\tilde{O}(\mathcal{M}_\alpha^n)$, where:

$$\mathcal{T}_\alpha = \left(\frac{1}{\alpha^\alpha \cdot (1-\alpha)^{1-\alpha}}\right)^{1/2} \quad \text{and} \quad \mathcal{M}_\alpha = \left(\frac{1}{\alpha^\alpha \cdot (1-\alpha)^{1-\alpha}}\right)^{1/4}.$$

2.2 A more practical version of Shamir and Schroeppel algorithm

In practical implementations, the need to use balanced trees or similar data structures is cumbersome. As a consequence, we would like to avoid this altogether. In order to do this, we can use a variation of an algorithm presented in [2] to solve the problem of finding 4 elements from 4 distinct lists with bitwise sum equal to 0.

Let us start by describing this variation for an ordinary knapsack, with no auxilliary information about the number of knapsack elements appearing in the decomposition of S. We start by choosing a number M near $2^{n/4}$. We define $\mathcal{S}^{(1)}$, $\mathcal{S}^{(2)}$, $\mathcal{S}_L^{(1)}$ and $\mathcal{S}_R^{(1)}$ as previously. We also define $\mathcal{S}_L^{(2)}$ and $\mathcal{S}_R^{(2)}$ in the obvious manner. With these notations, solving the knapsack problem amounts to finding four elements $\sigma_L^{(1)}$, $\sigma_R^{(1)}$, $\sigma_L^{(2)}$ and $\sigma_R^{(2)}$ in the corresponding sets such that:

$$\begin{aligned} S &= \sigma_L^{(1)} + \sigma_R^{(1)} + \sigma_L^{(2)} + \sigma_R^{(2)}. \\ \text{This implies that: } \sigma_L^{(1)} + \sigma_R^{(1)} &\equiv S - \sigma_L^{(2)} - \sigma_R^{(2)} \pmod{M}. \end{aligned}$$

Let σ_M denote this common middle value modulo M. Since these value is not known, the algorithm successively tries each of the M possibilities for σ_M.

[4] Indeed, if $\ell > n/2$, it suffices to replace S by $S' = \sum_{i=1}^{n} a_i - S$ and to solve this complementary knapsack which has $\ell < n/2$.

For each trial value of σ_M, we then construct the set of all sums $\sigma_L^{(1)} + \sigma_R^{(1)}$ congruent to σ_M modulo M. This is easily done is we sort the set $\mathcal{S}_R^{(1)}$ by ascending (or descending) values modulo M. Indeed, in this case, it suffices for each $\sigma_L^{(1)}$ in $\mathcal{S}_L^{(1)}$ to search the value $\sigma_M - \sigma_L^{(1)}$ in $\mathcal{S}_R^{(1)}$. With this method, building the intermediate set for a given value of σ_M costs $\tilde{O}(\max(|\mathcal{S}_L^{(1)}|, |\mathcal{S}_R^{(1)}|, |\mathcal{S}_L^{(1)}| \cdot |\mathcal{S}_R^{(1)}|/M) = \tilde{O}(2^{n/4})$. We proceed similarly to construct the sums $\sigma_L^{(2)} + \sigma_R^{(2)}$ congruent to $S - \sigma_M$ modulo M. The solution(s) to the knapsack problem are found by finding collisions between these lists of reduced size.

Taking into account the loop on σ_M, we obtain an algorithm with time complexity $\tilde{O}(2^{n/2})$ and memory complexity $\tilde{O}(2^{n/4})$. The same idea can also be used for unbalanced knapsacks, yielding the same time and memory complexities as in Section 2.1.

A high bit version. Instead of using modular values to classify the values $\sigma_L^{(1)} + \sigma_R^{(1)}$ and $S - \sigma_L^{(2)} - \sigma_R^{(2)}$, another option is to look at the value of the $n/4$ higher bits. Depending on the precise context, this option might be more practical than the modular version. In the implementation presented in Section 4, we make use of both versions.

3 The new algorithms

3.1 Basic principle

In this section, we assume that we want to solve a generic knapsack problem on n-elements, where n is a multiple of 4. We start from the basic knapsack equation:

$$S = \sum_{i=1}^{n} \epsilon_i a_i.$$

For simplicity of exposition[5], we also assume that $\sum_{i=1}^{n} \epsilon_i = n/2$, i.e., that S is a sum of precisely $n/2$ knapsack elements.

We define the set $\mathcal{S}_{n/4}$ as the set of all partials sums of $n/4$ knapsack elements. Clearly, there exists pairs (σ_1, σ_2) of elements of $\mathcal{S}_{n/4}$ such that $S = \sigma_1 + \sigma_2$. In fact, there exist many such pairs, corresponding to all the possible decompositions of the set of $n/2$ elements appearing in S into two subsets of size $n/4$. The number of such decompositions is given by the binomial $\binom{n/2}{n/4}$.

The basic idea that underlies all algorithms presented in this paper is to focus on a small part on $\mathcal{S}_{n/4}$, in order to discover one of these many solutions. We start by choosing an integer M near $\binom{n/2}{n/4}$ and a random element R modulo M. With some constant probability, there exists a decomposition of S into $\sigma_1 + \sigma_2$, such that $\sigma_1 \equiv R \pmod{M}$ and $\sigma_2 \equiv S - R \pmod{M}$. To find such a decomposition, it suffices to construct the two subsets of $\mathcal{S}_{n/4}$ containing elements

[5] When the number of elements in the sum is not precisely $n/2$, we may (by running the algorithm on S and $\sum_i a_i - S$) assume that is at most $n/2$. In this case, it suffices to replace $\mathcal{S}_{n/4}$ by the set of partial sums of at most $n/4$ elements.

respectively congruent to R and $S - R$ modulo M. The expected size of each these subsets is:

$$\frac{\binom{n}{n/4}}{M} \approx \frac{\binom{n}{n/4}}{\binom{n/2}{n/4}} \approx 2^{0.311\,n}.$$

Once these two subsets $\mathcal{S}_{n/4}^{(1)}$ and $\mathcal{S}_{n/4}^{(2)}$ are constructed, we need to find a collision between σ_1 and $S - \sigma_2$, with σ_1 in $\mathcal{S}_{n/4}^{(1)}$ and σ_2 in $\mathcal{S}_{n/4}^{(2)}$. Clearly, using a classical sort and match method, this can be done in time $\tilde{O}(2^{0.311\,n})$.

As a consequence, we can hope to construct an algorithm with overall complexity $\tilde{O}(2^{0.311\,n})$ for solving generic knapsacks, assuming that we can construct the sets $\mathcal{S}_{n/4}^{(1)}$ and $\mathcal{S}_{n/4}^{(2)}$ quickly enough. The rest of this paper shows how this can be achieved and also tries to minimize the required amount of memory.

Application to unbalanced knapsacks. Clearly, the same idea can be applied to unbalanced knapsack involving $\ell = \alpha n$ elements in the decomposition of S. In that case, S can be split into two parts, in approximately $2^{\alpha n}$ ways. The involved set of partial sums, now is $\mathcal{S}_{\ell/2}$ and we can restrict ourselves to subsets of size near:

$$\frac{\binom{n}{\ell/2}}{\binom{\ell}{\ell/2}} \approx \left(\frac{2}{\alpha^{\alpha/2} \cdot (2-\alpha)^{(2-\alpha)/2}}\right)^n \cdot 2^{-\alpha n}.$$

However, contrarily to what we have in Section 2.1, here it is better to assume that $\ell > n/2$. Indeed, when $\alpha > 1/2$ it is possible to decompose S into a much larger number of ways and obtain a better complexity. Alternatively, in order to preserve the convention $\alpha \leq 1/2$, we can rewrite the expected complexity as:

$$\frac{\binom{n}{\ell/2}}{\binom{n-\ell}{n/2-\ell/2}} \approx \left(\frac{2}{(1-\alpha)^{(1-\alpha)/2} \cdot (1+\alpha)^{(1+\alpha)/2}}\right)^n \cdot 2^{(\alpha-1)n}.$$

3.2 First algorithm

We first present a reasonably simple algorithm, which achieves running time $\tilde{O}(2^{0.420\,n})$ using $\tilde{O}(2^{0.210\,n})$ memory units. For simplicity of exposition, we need to assume here that n is a multiple of 32. Instead of considering decompositions of S into two sub-sums, we consider decompositions into four parts and write:

$$S = \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4,$$

where each σ_i belongs to the set $\mathcal{S}_{n/8}$ of all partials sums of $n/8$ knapsack elements. The number of such decompositions is given by the multinomial:

$$\binom{n/2}{n/8\ n/8\ n/8\ n/8} = \frac{(n/2)!}{(n/8)!^4} \approx 2^n.$$

We now choose an integer M near $2^{n/3}$ and three random elements R_1, R_2 and R_3 modulo M. With some constant probability, there exists a decomposition of S into $\sigma_1+\sigma_2+\sigma_3+\sigma_4$, such that $\sigma_1 \equiv R_1 \pmod{M}$, $\sigma_2 \equiv R_2 \pmod{M}$, $\sigma_3 \equiv R_3 \pmod{M}$ and $\sigma_4 \equiv S - R_1 - R_2 - R_3 \pmod{M}$. To find such a decomposition, we start by constructing the four subsets of $\mathcal{S}_{n/8}$ containing elements respectively congruent to R_1, R_2, R_3 and $S - R_1 - R_2 - R_3$ modulo M. We denote these subsets by $\mathcal{S}^{(1)}_{n/8}$, $\mathcal{S}^{(2)}_{n/8}$, $\mathcal{S}^{(3)}_{n/8}$ and $\mathcal{S}^{(4)}_{n/8}$.

Constructing the subsets. To construct each of the subsets $\mathcal{S}^{(1)}_{n/8}$, $\mathcal{S}^{(2)}_{n/8}$, $\mathcal{S}^{(3)}_{n/8}$ and $\mathcal{S}^{(4)}_{n/8}$, we can use the unbalanced version of the practical version of Shamir-Schroeppel algorithm as described in Section 2.2. Each subset is obtained by computing the sums of four elements, each of these elements being a sum of $n/32$ among $n/4$. The lists that contain these elements are of size $\binom{n/4}{n/32} \approx 2^{0.136\,n}$.

However, there is a small difficulty that appears at that point. Indeed, we wish to construct elements that satisfy a modular constraint, for example each σ_1 in $\mathcal{S}^{(1)}_{n/8}$ should be congruent to R_1 modulo M. This differs from the description of Shamir-Schroepel algorithm and its variation, which consider exact sums of the ring of integer. Thankfully, assuming that all values modulo M are represented by an integer in the range $[0, M[$, a sum of four values that is congruent to R_1 modulo M is necessarily equal to R_1, $R_1 + M$, $R_1 + 2M$ or $R_1 + 3M$. As a consequence, by running four consecutive instances of the Shamir-Schroeppel algorithm, it is possible to construct each of the sets $\mathcal{S}^{(i)}_{n/8}$ using time $\tilde{O}(2^{0.272\,n})$ and memory $\tilde{O}(2^{0.136\,n})$. Both the time and memory requirements of this stage are negligible compared to the complexity of the final stage.

Recovering the desired solution. Once the subsets $\mathcal{S}^{(1)}_{n/8}$, $\mathcal{S}^{(2)}_{n/8}$, $\mathcal{S}^{(3)}_{n/8}$ and $\mathcal{S}^{(4)}_{n/8}$ have been constructed, it suffices to directly apply the final phase of Shamir and Schroeppel algorithm, preferably using the practical version of Section 2.2. This allows us to find any existing collision between these two lists of values. The expected decomposition of S into $\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4$ is clearly obtained as a collision of this form.

As usual with Shamir and Schroeppel algorithm, with lists of size N, the complexity of this phase is $\tilde{O}(N^2)$ and the memory requirement $\tilde{O}(N)$. For this particular application, we have:

$$N \approx \frac{\binom{n}{n/8}}{M} \approx \frac{2^{0.544\,n}}{2^{n/3}} \approx 2^{0.210\,n}.$$

Finally, since this phase dominates the complexity, we conclude that the overall complexity of our first algorithm is $\tilde{O}(2^{0.420\,n})$, using $\tilde{O}(2^{0.210\,n})$ memory units.

Complexity for unbalanced knapsacks. This algorithm can easily by extended to unbalanced knapsacks of weigth ℓ. Writing $\ell = \alpha\, n$, we find that the number of different decompositions of the sought after solution is:

$$\binom{\ell}{\ell/4\ \ell/4\ \ell/4\ \ell/4} = \frac{\ell\,!}{(\ell/4)!^4} \approx 4^{\alpha\, n}.$$

We thus choose $M_\alpha \approx 4^{\alpha\, n/3}$. During the construction of the subsets $\mathcal{S}^{(i)}_{\ell/4}$, we take sums from elements make from $\ell/16$ elements among $n/4$. They have size:

$$N_\alpha^{(0)} = \binom{n/4}{\ell/16} \approx \left(\frac{4^{1/4}}{(\alpha^\alpha \cdot (4-\alpha)^{4-\alpha})^{1/16}}\right)^n .$$

The time and memory complexities of the construction phase are $\tilde{O}({N_\alpha^{(0)}}^2)$ and $\tilde{O}(N_\alpha^{(0)})$.

During the recovery phase, we have sets of size:

$$N_\alpha \approx \frac{\binom{n}{\ell/4}}{M_\alpha} \approx \left(\frac{4}{\alpha^{\alpha/4} \cdot (4-\alpha)^{(4-\alpha)/4} \cdot 2^{2\alpha/3}}\right)^n .$$

Thus, we obtain an algorithm with time complexity $\tilde{O}(N_\alpha^2)$ and memory complexity $\tilde{O}(N_\alpha)$. Indeed, for any value of α in the range $]0, 1/2]$, we have $N_\alpha > N_\alpha^{(0)}$.

3.3 An improved algorithm

In this section, we modify the simplified algorithm in order to improve it and be closer to the expected $\tilde{O}(2^{0.311\, n})$ from Section 3.1, without increasing the memory complexity too much. In order to do this, we start by relaxing the constraints on the sets $\mathcal{S}^{(1)}_{n/8}$, $\mathcal{S}^{(2)}_{n/8}$, $\mathcal{S}^{(3)}_{n/8}$ and $\mathcal{S}^{(4)}_{n/8}$. More precisely, instead of prescribing there values modulo a number M near $2^{n/3}$, we now choose a smaller M near $2^{\beta n}$, with $\beta < 1/3$. As a consequence, we now expect $2^{(1-3\beta)n}$ different four-tuples $(\sigma_1, \sigma_2, \sigma_3, \sigma_4)$ in $\mathcal{S}^{(1)}_{n/8} \times \mathcal{S}^{(2)}_{n/8} \times \mathcal{S}^{(3)}_{n/8} \times \mathcal{S}^{(4)}_{n/8}$, with sum equal to the target S.

Clearly, this change does not modify the size of the smaller lists used to constuct the sets $\mathcal{S}^{(i)}_{n/8}$. Thus, the memory complexity of the construction step remains unchanged. However, the time complexity of this phase may become higher. Indeed the resulting sets $\mathcal{S}^{(i)}_{n/8}$ can now be much larger. The sets have size $\binom{n}{n/8} \cdot 2^{-\beta n} \approx 2^{(0.544-\beta)n}$. As a consequence, the time complexity becomes $\tilde{O}(\max(2^{(0.544-\beta)n}, 2^{0.272\, n}))$.

Once the sets $\mathcal{S}^{(1)}_{n/8}$, $\mathcal{S}^{(2)}_{n/8}$, $\mathcal{S}^{(3)}_{n/8}$ and $\mathcal{S}^{(4)}_{n/8}$ have been constructed, we again need to run Shamir-Schroeppel method. We have four list of size $\tilde{O}(2^{(0.544-\beta)n})$, thus one might expect a time proportional to the square of the lists size. However, recalling that the sets contain about $2^{(1-3\beta)n}$ solutions to the system, it suffices to run Shamir-Schroeppel on a fraction of the middle values to reach a solution. This fraction is approximately $2^{(3\beta-1)n}$. As a consequence, we obtain an algorithm that runs in time $\tilde{O}(\max(2^{(0.088+\beta)n}, 2^{(0.544-\beta)n}, 2^{0.272\, n}))$ using $\tilde{O}(2^{(0.544-\beta)n})$ units of memory.

We can minimize the time by taking $\beta \approx 0.228$ and obtain an algorithm with time complexity $\tilde{O}(2^{0.315\, n})$ and memory complexity $\tilde{O}(2^{0.315\, n})$. Of course, different time-memory tradeoffs are also possible for this algorithm. More precisely, for any $0.228 \le \beta \le 1/3$ we obtain a different tradeoff. If we let $\mathcal{T}_\beta$ and $\mathcal{M}_\beta$ denote the corresponding time and memory complexity, the key property is that the product $\mathcal{T}_\beta \cdot \mathcal{M}_\beta$ remains constant as β varies in the above stated range.

Complexity for unbalanced knapsacks. As before, this algorithm can easily by extended to unbalanced knapsacks of weigth ℓ. Again writing $\ell = \alpha\, n$ and taking M_α near $2^{\beta\, n}$, we obtain new sizes for the involved sets. As previously, we have:

$$N_\alpha^{(0)} = \binom{n/4}{\ell/16} \approx \left(\frac{4^{1/4}}{(\alpha^\alpha \cdot (4-\alpha)^{4-\alpha})^{1/16}}\right)^n.$$

Moreover, completing our previous notations by the parameter $0 \le \beta \le 2\alpha/3$, we find that:

$$N_{\alpha,\beta} \approx \frac{\binom{n}{\ell/4}}{M_\alpha} \approx \left(\frac{4}{\alpha^{\alpha/4} \cdot (4-\alpha)^{(4-\alpha)/4} \cdot 2^\beta}\right)^n.$$

The time complexity now is $\tilde{O}(\max({N^{(0)}}^2, N_{\alpha,\beta}, N_{\alpha,\beta}^2 \cdot 2^{(3\beta-2\alpha)n})$ and the memory complexity becomes $\tilde{O}(N_{\alpha,\beta})$, assuming $N_{\alpha,\beta} \ge N^{(0)}$ which holds for any reasonable choice of parameters.

In particular, we can minimize the running time by choosing:

$$\beta = \max\left(0, \alpha - \frac{1}{2}\log_2\left(\frac{4}{\alpha^{\alpha/4} \cdot (4-\alpha)^{(4-\alpha)/4}}\right)\right)$$

With this choice, the time complexity is $\tilde{O}(2^{C_\alpha n})$ and the memory complexity $\tilde{O}(2^{C_\alpha n})$), where:

$$C_\alpha = \max\left(\frac{3}{2}\log_2\left(\frac{4}{\alpha^{\alpha/4} \cdot (4-\alpha)^{(4-\alpha)/4}}\right) - \alpha, 2\log_2\left(\frac{4}{\alpha^{\alpha/4} \cdot (4-\alpha)^{(4-\alpha)/4}}\right) - 2\alpha\right).$$

Of course, as in the balanced case, different time-memory tradeoffs are also possible.

Knapsacks with multiple solutions. Note that nothing prevents the above algorithm finding a large fraction of the solutions for knapsacks with many solutions. However, in that case, we need to take some additional precautions. First, we must take care to remove duplicate solutions when they occur. Second, we should remember that, if the number of solutions is extremely large it can become the dominating factor of the time complexity.

It is also important to remark that, for an application that would require all the solutions of the knapsack, it would be necessary to increase the running time. The reason is that this algorithm is probabilistic and that the probability of missing any given solution decreases exponentially as a function of the running time. Of course, when there is a large number N_{sol} of solutions, the probability of missing at least one is multiplied by N_{sol}. To balance this, we need to increase the running time by a factor of $\log(N_{\mathrm{sol}})$.

3.4 A complete algorithm for the balanced case

Despite the fact that the algorithms from Sections 3.2 and 3.3 outperform Shamir-Schroeppel method, they do not achieve the complexity expected from Section 3.1. The algorithm from Section 3.3 comes close but it requires more memory than we would expect.

In order to further reduce the complexity, we need to follow more closely the basic idea from Section 3.3. More precisely, we need to write $S = \sigma_1 + \sigma_2$ and constrain σ_1 enough to lower the number of expected solutions to 1. Once again, we assume, for simplicity, that S is a sum of exactly $n/2$ elements of the knapsacks.

Of course, this can be achieved by keeping one solution out of $2^{n/2}$. One option would be to choose a modulus M close to $2^{n/2}$ and fix some random value for σ_1. However, in order to have more flexibility, we choose $\gamma \geq 1/2$, consider a larger modulus M close to $2^{\gamma n}$ and allow $2^{(\gamma - 1/2)n}$ different random values for σ_1.

For each of these $2^{(\gamma-1/2)n}$ random values, say R, we need to compute the list of all solutions to the partial knapsack $\sigma_1 = R \pmod{M}$, the list of all solutions to $\sigma_2 = S - R \pmod{M}$ and finally to search for a collision between the complete integer values of σ_1 and $S - \sigma_2$.

Clearly, the list of values σ_1 (or σ_2) can be constructed by solving a modular knapsack problem involving about $n/4$ values chosen among n. With the usual idea of randomizing the order of knapsack elements and running the algorithm several times, we may assume that σ_1 is the sum of precisely $n/4$ values. Thus, we can obtain the list of all σ_1 values by running an integer knapsack algorithm $n/4$ times, for each of the following targets: R, $R+M$, $R+2M$, $\ldots$, $R+(n/4-1)M$. Using the algorithm from Section 3.3 with $\alpha = 1/4$ and the corresponding choice $\beta \approx 0.081$, this can be done using memory $\tilde{O}(2^{0.256\,n})$. Since the number of expected solutions is $\tilde{O}(2^{(0.811-\gamma)n})$, the running time is $\tilde{O}(\max(2^{0.256\,n}, 2^{(0.811-\gamma)n}))$. Choosing $\gamma \approx 0.555$, we balance the memory and running time of the subroutine (resp. $\tilde{O}(2^{0.256\,n})$ and $\tilde{O}(2^{0.256\,n})$).

The total running time is obtained by multiplying the time of the subroutine by the number of consecutive runs, i.e. $2^{(\gamma-1/2)n}$. As a consequence, the complete algorithm runs using time $\tilde{O}(2^{0.311\,n})$ and memory $\tilde{O}(2^{0.256\,n})$. In terms of memory, this is just slightly worse than Shamir-Schroeppel.

Unbalanced knapsacks. Note that this algorithm does not behave well for unbalanced knapsacks. At the time of writing, we have not been able to find a generalization for unbalanced knapsacks that achieves the expected running time and remains good in terms of memory.

4 A practical experiment

In order to make sure that our new algorithms perform well in practice, we have benchmarked their performance by using a typical hard knapsack problem. We constructed it using 96 random elements of 96 bits each and then built the target S as the sum of 48 of these elements. For this knapsack instance, we now compare the performance of Shamir-Schroeppel (practical version), of the algorithm[6] from Section 3.3 .

Shamir-Schroeppel algorithm. Concerning the implementation of Shamir-Schroeppel algorithm, we need to distinguish between two cases. Either we are given a good decomposition of the set of indices into four quarters, each containing half zeroes and half ones, or we are not. In the

[6] We skip the algorithm from Section 3.2 because the program is identical to the algorithm of Section 3.3 and relies on different less efficient parameters.

first case, we can reduce the size of the initial small lists to $\binom{24}{12} = 2\,704\,156$. In second case, two options are possible: we can run the previous approach on randomized decompositions until a good is found, which requires about 120 executions; or we can start from small lists of size $2^{24} = 16\,777\,216$.

For the first case, we used the variation of Shamir-Schroeppel presented in Section 2.2 with a prime modulus $M = 2\,704\,157$. For lack of time, we could not run the algorithm completely. From the performance obtained by running on a subset of the possible values modulo M, we estimate the total running time to 37 days one a single Intel core 2 duo at 2.66 Ghz. For the second case, it turns out that, despite higher memory requirements, the second option is the faster one and would require about 1 500 days on the same machine (instead of the 4 400 days for the first option).

In our implementation of Shamir-Schroeppel algorithm, in order to optimize both the running time and the memory usage, we store the sets $\mathcal{S}^{(1)}$, $\mathcal{S}^{(2)}$, $\mathcal{S}_L^{(1)}$ and $\mathcal{S}_R^{(1)}$ without keeping track of the decomposition of their elements into a sum of knapsack elements. This means that the program detects correct middle values modulo M but cannot recover the knapsack decomposition immediately. Instead, we need to rerun a version that stores the full description on the correct middle values. All in all, this trick allows our program to run twice faster and to use only 300 Mbytes of memory with initial lists of size $\binom{24}{12}$ and 1.8 Gbytes with initial lists of size 2^{24}.

Our improved algorithm. As with the Shamir-Schroeppel algorithm, we need to distinguish between two cases, depending whether or not a good decomposition into four balanced quarters is initially known. When it is the case, our implementation recovers the correct solution in slightly less than an hour on the same computer. When such a decomposition is not known in advance, we use the same approach on randomized decompositions and find a solution in about 5 days. The parameters we use in the implementation are the following:

- For the main modulus that define the random values R_1, R_2 and R_3, we take $M = 1\,200\,007$.
- Concerning the Shamir-Schroeppel subalgorithms, we need to assemble four elements coming from small lists of 2 024 elements, that sum to one of the four specified values modulo M, i.e. to R_1, R_2, R_3 or $S - R_1 - R_2 - R_3$. We use a variation of Shamir-Schroeppel that works on the 12 highest bits of the middle value. The resulting lists have size near 14 000 000
- For the final merging of the four obtained lists, we use a modular version of Shamir-Schroeppel algorithm with modulus 2 000 003.

In this implementation, we cannot use the same trick as in Shamir-Schroeppel and forgot the description of each element into a sum of knapsack elements. Indeed, each we merge two elements together, it is important to check that they do not share a common knapsack element. Otherwise, we obtain many fake decompositions of the target, which greatly lower the performance of the program. All in all, the program uses approximately 2.7 Gbytes of memory.

Our complete algorithm. Before giving detailed parameters and running times, we would like to mention that in out implementation, the small lists that occur are so small that we replaced the innermost Shamir-Schroeppel algorithm running each small knapsack (of weight 6 in 96) by a basic birthday paradox method. One consequence of this is that we not longer need a decomposition of the knapsack into four balanced quarters. Instead, two balanced halves are enough. This means that when such a decomposition is not given, we only need to run the algorithm an average of 6.2 times to find a correct decomposition. The parameters we use in the implementation are the following:

- For the main modulus that define the random value R we take $M = (2^{23}+9)\cdot 1009 \cdot 50\,021$.
- Concerning the two subinstances of the improved algorithm that appear within the complete algorithm, we define the value R_1, R_2, R_3 modulo 50 021. As a consequence, the innermost birthday paradox method is matching values modulo 50 021.
- The assembling phase of the two subinstances uses the modulus 1009 to define its middle values.

This specific choice of having a composite values for M whose factors are used in the two instances of the improved algorithm allowed us to factor a large part of the computations involved in these two subinstances. In particular, we were able to use the same values of R_1, R_2, R_3 and $S' - R_1 - R_2 - R_3$ for both instances by choosing only values of R such that $R = S - R$ modulo $1009 \cdot 50\,021$. We also take care to optimize the running time each trial by halting the subalgorithms early enough to prevent them from producing the same solutions too many times. This lowers the probability of success of individual trails but improves the overall running time.

Another noteworthy trick is that, instead of trying all the targets of the form R, $R + M$, $R + 2M$, ..., $R + (n/4 - 1)M$, we only consider a small number of targets around the $R + (n/8)\,M$. Since a sum of $n/4$ random numbers modulo M as an average of $(n/8)\,M$, this does not lower the probability of success much and gives a nice speedup.

Using these practical tricks, our implementation of the complete algorithm runs reasonnably fast. We have not made enough experiments to get a precise average running time, however, on two different starting points, it ran once in about 6 hours and once in about 12 hours (on the same processor), given a good decomposition of the knapsack into two halves. Without this decomposition and taking the higher of the two estimates, it would require about 3 days to find the decomposition. Where memory is concerned, the program uses less than 500 Mbytes.

5 Possible extensions and applications

The algorithmic techniques presented in this paper can be applied to more than ordinary knapsacks, we give here a list of different problems were the same algorithms can also be used.

Approximate knapsack problems. A first problem we can consider is to find approximate solutions to knapsack problems. More precisely, given a knapsack a_1, ..., a_n and a target S, we try to write:

$$S = \sum_{i=1}^{n} \epsilon_i \, a_i + \delta,$$

where δ is small, i.e. belongs to the range $[-B, B]$ for a specified bound B.

This problem can be solved by transforming it into a knapsack problem with several targets. Define a new knapsack $b_1, \ldots, b_n$ where b_i is the closest integer to a_i/B and let S' be the closest integer to S/B. To solve the original problem, it now suffices to find solutions to the new knapsack, with targets $S'-(n/2), \ldots, S'+(n/2)$ when n is even or with targets $S'-((n+1)/2)$, $\ldots$, $S'+((n+1)/2)$ when n is odd.

Modular knapsack problems. In the course of developping our algorithms, we have seen that modular knapsacks can be solved simply by lifting to the integers and using multiple targets. If the modulus is M and the modular target is T, it suffices to consider the integet targets T, $T+M$, $\ldots$, $T+nM$.

Vectorial knapsack problems. Another option is to consider knapsacks whose elements are vectors of integers and where the target is a vector. Without going into the details, it is clear that this is not going to be a problem for our algorithms. In fact, the decomposition into separate components can even make things easier. Indeed, if the individual components are of the right size, they can be use as a replacement for the modular criteria that determine whether we keep or remove partial sums.

Knapsacks with ϵ_i in $\{-1, 0, 1\}$. In this case, we can applied similar methods. However, we obtain different bounds, since the number of different representations of a given solution is no longer the same. For example, going back to the theoretical bound, a solution with ℓ coefficients equal to -1 and ℓ coefficients equal to 1 can be split into two parts each containing $\ell/2$ coefficients of both types in $\binom{\ell}{\ell/2}^2$. ways.

In order to show the applicability of our techniques in this case, we recompute the time complexities of the Shamir-Schroeppel algorithm and of our improved algorithm assuming that $\ell = \alpha \cdot n$. For Shamir-Schroeppel, the time complexity is the same as for the basic birthday algorithm, i.e, it is equal to:

$$\binom{n/2}{\ell/2 \;\ell/2\; (n/2-\ell)} \approx \left(\frac{1}{\alpha^{\alpha} \cdot (1-2\,\alpha)^{(1-2\alpha)/2}} \right)^{n}.$$

For the improved algorithm, we need to update the values of our parameters, we first find that the number of different decomposition of a given solution into 4 parts is:

$$\binom{\ell}{\ell/4 \;\ell/4\; \ell/4 \;\ell/4}^2 \approx 4^{2\alpha\, n}.$$

We also can recompute:

$$N_{\alpha}^{(0)} = \binom{n/4}{\ell/16 \;\ell/16\; (n/4-\ell/8)} \approx \left(\frac{2^{1/4}}{((\alpha/2)^{\alpha} \cdot (2-\alpha)^{2-\alpha})^{1/8}} \right)^{n}.$$

Moreover, under the assumption that $0 \leq \beta \leq 4\alpha/3$, we now define:

$$N_{\alpha,\beta} \approx \frac{\binom{n}{\ell/4\ \ell/4\ (n-\ell/2)}}{M_\alpha} \approx \left(\frac{2}{(\alpha/2)^{\alpha/2} \cdot (2-\alpha)^{(2-\alpha)/2} \cdot 2^\beta}\right)^n.$$

The time complexity thus becomes $\tilde{O}(\max({N^{(0)}}^2, N_{\alpha,\beta}, N_{\alpha,\beta}^2 \cdot 2^{(3\beta-4\alpha)n})$.

Generalized birthday generalization. Since the paper of Wagner [17], it is well-known that when solving problems involving sums of elements from several lists, it is possible to obtain a much faster algorithm when a single solution out of many is sought. In some sense, the algorithms from the present paper are a new developpement of that idea. Indeed, we are looking for one representation out of many for a given solution. With this restatement, it is clear that further improvements can be obtained if, in addition, we seek one solution among many. Putting everything together, we need to seek a single representation among a very large number obtained by multiplying the number of solutions by the average number of representations for each solution.

However, note that this approach does not work for all problems where the generalized birthday paradox can be applied. It can only improve the efficiency when the constraints on the acceptable solutions are such that allowing multiple representations really add new options. Indeed, if one does not take care, there is a risk of counting each solution several times.

Combination of the above and possible applications. In fact, it is even possible to address any combination of the above. As a consequence, this algorithm can be a very useful cryptanalytic tool. For example, the NTRU cryptosystem can be seen as an unbalanced, approximate modular vector knapsack. However, it has been shown in [8] that it is best to attack this cryptosystem by using a mix of lattice reduction and knapsack-like algorithms. As a consequence, deriving new bounds for attacking NTRU would require a complex analysis, which is out of scope for the present paper.

In the same vein, Gentry's fully homomorphic scheme [6], also needs to be studied with our new algorithm in mind.

Another possible application is the SWIFFT hash function [11]. Indeed, as explained by the authors, this hash function can be rewritten as a vectorial modular knapsack. Moreover, searching for a colision on the hash function can be interpreted as a knapsack with coefficients in $\{-1, 0, 1\}$. In [11], the application of generalized birthday algorithms to cryptanalyze SWIFFT is already studied. It would be interesting to re-evaluated the security of SWIFFT with respect to our algorithm. In order to show that the attack of [11] based on Wagner's generalized birthday algorithm can be improved upon, we give a first improvement in appendix B.

Note that, in all cases, we do not affect the asymptotic security of all the scheme, indeed, an algorithm with complexity $2^{0.311n}$ remains exponential. However, depending in the initial designers hypothesis, recommended practical parameters may need to be enlarged. For the special case of NTRU, it can be seen that in [7] that the estimates are conservative enough not to be affected by the our algorithms.

6 Conclusion

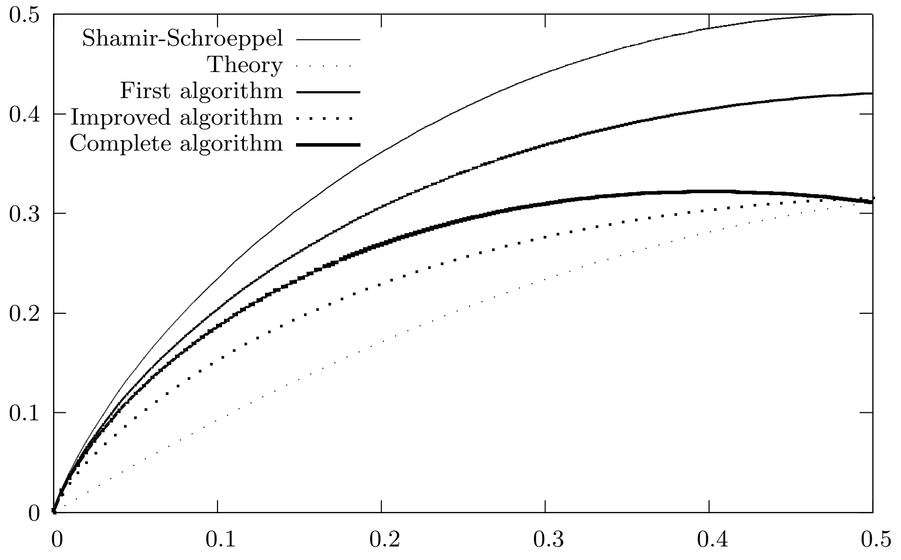
In this paper, we have proposed new algorithms to solve the knapsack problem and other related problems, which improve on the current state of the art. In particular, for the knapsack problem itself, this improves the 28-year old algorithm of Shamir and Schroeppel and gives a positive answer to the question posed in the Open Problem Garden [4] about knapsack problems: "Is there an algorithm that runs in time $2^{n/3}$?". Since our algorithm is probabilistic, an interesting open problem is to give a fast deterministic algorithm for the same task.

References

1. Miklós Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In *30th ACM STOC*, pages 10–19, Dallas, Texas, USA, May 23–26, 1998. ACM Press.
2. Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 209–221, Amsterdam, The Netherlands, April 28–May 2, 2002. Springer-Verlag, Berlin, Germany.
3. Matthijs J. Coster, Antoine Joux, Brian A. LaMacchia, Andrew M. Odlyzko, Claus-Peter Schnorr, and Jacques Stern. Improved low-density subset sum algorithms. *Computational Complexity*, 2:111–128, 1992.
4. Open problem garden. http://garden.irmacs.sfu.ca.
5. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.
6. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178, Bethesda, MD, USA, may 2009. ACM Press.
7. Philip S. Hirschorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches. In *Applied cryptography and network security*, volume 5536 of *LNCS*, pages 437–455. Springer-Verlag, Berlin, Germany, 2009.
8. Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169, Santa Barbara, CA, USA, August 19–23, 2007. Springer-Verlag, Berlin, Germany.
9. Jeffrey C. Lagarias and Andrew. M. Odlyzko. Solving low-density subset sum problems. *J. Assoc. Comp. Mach.*, 32(1):229–246, 1985.
10. Arjen K. Lenstra, Hendrick W. Lenstra, Jr., and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
11. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In Kaisa Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 54–72, Lausanne, Switzerland, February 10–13, 2008. Springer-Verlag, Berlin, Germany.
12. Ralph Merkle and Martin Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Trans. Information Theory*, 24(5):525–530, 1978.
13. Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.
14. Richard Schroeppel and Adi Shamir. A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.
15. Andrew Shallue. An improved multi-set algorithm for the dense subset sum problem. In *Algorithmic Number Theory, 8th International Symposium*, volume 5011 of *LNCS*, pages 416–429. Springer-Verlag, Berlin, Germany, 2008.
16. Adi Shamir. A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 279–288, Santa Barbara, CA, USA, 1983. Plenum Press, New York, USA.
17. David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303, Santa Barbara, CA, USA, August 18–22, 2002. Springer-Verlag, Berlin, Germany.

A Graph of compared complexities

In the following figure, we present the complexity of the algorithms discussed in the paper as a function of the proportion of 1s in the knapsack. In particular, this graph shows that the complete algorithm does not well for unbalanced knapsacks.



B On the security of SWIFFT

Here, we detail the impact of the generalized algorithms of Section 5 on the SWIFFT hash function. In [11], Section 5.2, paragraph "Generalized Birthday Attack", this security is assessed by using Wagner's algorithm on a modular vectorial knapsack with entries $\{-1, 0, 1\}$. This knapsack contains 1024 vectors of 64 elements modulo 257. The principle of the attack consists in breaking the 1024 vectors into 16 groups of 64 vectors each. For each group, the attack builds a list of $3^{64} \approx 2^{102}$ different partial sums. Finally Wagner's algorithm is used on these lists. This gives an algorithm with time at least 2^{106} and memory at least 2^{102}.

In order to show that this algorithm can be improved, we give a simplified version of our algorithms that lowers the time to approximately 2^{99} and the memory to 2^{96}. The idea is to start by building arbitary sums of the 1024 elements with a fraction of theri coordinates already equal to 0 modulo 257 and with coefficients in $\{0, 1\}$ instead of $\{-1, 0, 1\}$. Any collision between two such sums yields a collision of SWIFFT. To build the sums, we proceed as in [11] and split the 1024 vectors into 16 groups of 64 vectors. In each group, we construct the 2^{64} possible sums. Then, we assemble the groups by two forcing 4 coordinates to 0. After this phase, we obtain 8 lists of approximately 2^{96} elements. We assemble this list by pairs again, forcing 12 coordinates to 0 and obtain 4 lists of approximately 2^{96} elements. After two more assembling, we obtain a list of 2^{96} sums with a total of 40 coordinates equal to zero. A simple birthday paradox argument shows that we can find a collision between the 24 remaining coordinates. After subtracting the two expressions, we obtain the zero vector as a sum of the 1024 vectors with $\{-1, 0, 1\}$ coefficients, i.e. a collision on SWIFFT.

A more detailed analysis of the parameters of SWIFFT is still required to determine whether the collision attack can still be improved upon.