New Generic Algorithms for Hard Knapsacks

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, i.e., 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 31-year old algorithm by Schroeppel and Shamir 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 $\tilde{O}(2^{n/4})$ storage. We propose here two new algorithms which improve on this bound, finally lowering the running time down to either $\tilde{O}(2^{0.385\,n})$ or $\tilde{O}(2^{0.3113\,n})$ under a reasonable heuristic. We also demonstrate the practicality of these algorithms with an implementation.

1 Introduction

The 0–1 knapsack problem or subset sum 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 forms 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 $\epsilon = (\epsilon_1, \cdots, \epsilon_n)$ if at least one exists.

The decision knapsack problem is NP-complete (see [7]). It is also well-known that given access to an oracle that solves the decision problem, the computational problem can be solved using n calls to this oracle. 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 [18] 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 [23] 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 [15] and improved by Coster et al. [4] are a tool of choice for breaking many knapsack based cryptosystems. The density of a knapsack is defined as:

$$d = \frac{n}{\log_2(\max_i a_i)}.$$

More recently, Impagliazzo and Naor [13] introduced cryptographic schemes which are as secure as the subset sum problem. They classify knapsack problems according to their density. On the one hand, when $d < 1$ a given sum S can usually be inverted in a unique manner and these knapsacks can be used for encryption. On the other hand, when $d > 1$, most sums have many preimages and the knapsack can be used for hashing purposes. However, for encryption, the density cannot be too low, since the Lagarias-Odlyzko low-density attack can solve random knapsack problems 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[1] [16] or the BKZ algorithm of Schnorr [20]. The attack of [4] improves the low density condition to $d < 0.94$. For high density knapsacks, with $d > 1$ there is variation of these lattice-based attacks presented in [14] that finds collisions in mildly exponential time $O(2^{n/1000})$ using the same lattice reduction oracle.

However, for knapsacks with density close to 1, there is no effective lattice-based approach to solve the knapsack problem. As a consequence, in this case, we informally speak of *hard* knapsacks. Note that, it is proved in [13, Proposition 1.2], that density 1 is indeed the hardest case. For hard knapsacks, the state-of-the-art algorithm is due to Schroeppel and Shamir [21,22] 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 the basic birthday based algorithm on the knapsack problem introduced by Horowitz and Sahni [10], but much lower memory requirements. 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 Schroeppel and Shamir runs in time $\tilde{O}(2^{n/2})$ using $\tilde{O}(2^{n/4})$ bits of memory.

Since Wagner presented his generalized birthday algorithm in [25], 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. A similar idea was previously used by Camion and Patarin in [2] to attack the knapsack based hash function of [5]. In this paper, we introduce two new algorithms that improve upon the algorithm of Schroeppel and Shamir to solve knapsack problems. In some sense, our algorithms are a new development

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

of the generalized birthday algorithm. The main difference is that, instead of looking for one solution among many, we look for one of the many possible representations of a given solution.

The paper is organized as follows: In Section 2 we recall some background information on knapsacks, in Section 3 we briefly recall the algorithm of Schroeppel–Shamir and introduce a useful practical variant of this algorithm, in Section 4 we present our improved algorithms and in Section 5 we describe practical implementations on a knapsack with $n = 96$. Section 4 is divided into 3 subsections, in 4.1 we describe the basic idea that underlies our algorithm, in 4.2 we present a simple algorithm based on this idea and in 4.3 we give a heuristic improvement of this algorithm in the balanced case. Finally, in Section 6 we present several extensions and some possible applications of our new algorithms.

2 Background on Knapsacks

2.1 Modular Knapsacks

We speak of a modular knapsack problem when we want to solve:

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

where the integer M is the modulus.

Up to polynomial factors, solving modular knapsacks and knapsacks over the integers are equivalent. Any algorithm that realizes one task can be used to solve the other. In one direction, given a knapsack problem over the integers and an algorithm that solves any modular knapsack, it is clear that solving the problem modulo $M = \max(S, \sum_{i=1}^{n} a_i) + 1$ yields all integral solutions. In the other direction, assume that the modular knapsack $(a_1, \cdots, a_n)$ with target sum $S \bmod M$ is given by representative a_i of the classes of modular numbers in the range $[0, M-1]$. In this case, it is clear that any sum of at most n such numbers is in the range $[0, nM-1]$. As a consequence, if S is also represented in the range $[0, M-1]$, it suffices to solve n knapsack problems over the integers with targets S, $S+M$, $\ldots$, $S+(n-1)M$.

2.2 Random Knapsacks

Given two parameters n and D, we define a *random knapsack with solution* on n elements with prescribed density D as a knapsack randomly constructed using the following process:

- Let $B(n, D) = \lfloor 2^{n/D} \rfloor$.
- Choose each a_i (for i from 1 to n) uniformly at random in $[1, B(n, D)]$.
- Uniformly choose a random vector ϵ in $\{0, 1\}^n$ and let $S = \sum_{i=1}^{n} \epsilon_i \, a_i$.

Note that the computed density d of such a random knapsack differs from the prescribed density. However, as n tends to infinity, the two become arbitrarily close with overwhelming probability. In [4], it is shown that there exists a lattice based algorithm that solves all but an exponentially small fraction of random knapsacks with solution, when the prescribed density satisfies $D < 0.94$.

2.3 Unbalanced Knapsacks

The random knapsacks from above may have arbitrary values in $[0, n]$ for the weight $\sum_{i=1}^{n} \epsilon_i$ of the solution. Yet, most of the time, we expect a weight close to $n/2$. For various reasons, it is also useful to consider knapsacks with different weights. We define an *α-unbalanced random knapsack with solution* on n elements given α and the density D as follows:

- Let $B(n, D) = \lfloor 2^{n/D} \rfloor$.
- Choose each a_i (for i from 1 to n) uniformly at random in $[1, B(n, D)]$.
- Let $\ell = \lfloor \alpha n \rfloor$ and uniformly choose a random vector ϵ with exactly ℓ coordinates equal to 1, the rest being 0s, in the set of $\binom{n}{\ell}$ such vectors. Let $S = \sum_{i=1}^{n} \epsilon_i \, a_i$.

Unbalanced knapsacks are natural to consider, since they already appear in the lattice based algorithms of [15,4], where the value of α greatly impacts the densities that can be attacked. Moreover, in our algorithms, even when initially solving regular knapsacks, unbalanced knapsacks may appear in the course of the computations.

When dealing with balanced knapsacks with exactly half zeros and ones, we also use the above definition and speak of 1/2-unbalanced knapsacks.

2.4 Complementary Knapsacks

Given a knapsack $a_1, \ldots, a_n$ with target sum S, we define its *complementary knapsack* to be the knapsack that contains the same elements and has target sum $\sum_{i=1}^{n} a_i - S$. The solution ϵ of the original knapsack and ϵ' of the complementary knapsacks are related by:

$$\text{For all } i\text{:} \quad \epsilon_i + \epsilon'_i = 1.$$

Thus, solving either of the two knapsacks also yields the result of the other knapsack. Moreover, the weight ℓ and ℓ' are related by $\ell + \ell' = n$. In particular, if a knapsack is α-unbalanced, its complementary knapsack is $(1-\alpha)$-unbalanced. As a consequence, in any algorithm, we may assume without loss of generality that $\ell \leq \lfloor n/2 \rfloor$ (or that $\ell \geq \lceil n/2 \rceil$).

2.5 Asymptotic Values of Binomials

Where knapsacks are considered, binomial coefficients are frequently encountered, we recall that the binomial coefficient $\binom{n}{\ell}$ is the number of distinct choices of ℓ elements within a set of n elements. We have:

$$\binom{n}{\ell} = \frac{n!}{\ell! \cdot (n-\ell)!}.$$

We often need to obtain asymptotic approximation for binomials of the form $\binom{n}{\alpha\, n}$ (or $\binom{n}{\lfloor \alpha\, n \rfloor}$) for fixed values of α in $]0, 1[$. This is easily done by using Stirling's formula:

$$n! = (1 + o(1))\, \sqrt{2\pi n}\, \left(\frac{n}{e}\right)^n.$$

Ignoring polynomial factors in n, we find:

$$\binom{n}{\alpha\, n} = \tilde{O}\left(\left(\frac{1}{\alpha^\alpha \cdot (1-\alpha)^{1-\alpha}}\right)^n\right).$$

Many of the algorithms presented in this paper involve complexities of the form $\tilde{O}(2^{c\, n})$, where a constant c is obtained by taking the logarithm in basis 2 of numbers coming from asymptotic estimates of binomials. In this case, to improve the readability of the complexity, we choose a decimal approximation $c_0 > c$ of c. This would allow us to rewrite the complexity as $O(2^{c_0\, n})$ or even $o(2^{c_0\, n})$. However, we prefer to stick to $\tilde{O}(2^{c_0\, n})$. A typical example is the $\tilde{O}(2^{0.3113\, n})$ time complexity of our fastest algorithm, which stands for $\tilde{O}\left(\binom{n}{n/4} \cdot 2^{-n/2}\right)$.

2.6 Distribution of Random Knapsack Sums

In order to analyze the behavior of our algorithms, we need to use information about the distribution of modular sums of the form:

$$\sum_{i=1}^{n} a_i x_i \pmod{M},$$

for a random knapsack modulo M and for n-tuples $(x_1, \cdots, x_n) \in \mathcal{B}$, where $\mathcal{B}$ is an arbitrary set of n-dimensional vectors, with coordinates modulo M. We use the following important theorem [19, Theorem 3.2]:

Theorem 1. *For any set $\mathcal{B} \subset \mathbb{Z}_M^n$, the identity:*

$$\frac{1}{M^n} \sum_{(a_1,\cdots,a_n)\in\mathbb{Z}_M^n} \sum_{c\in\mathbb{Z}_M} \left(P_{a_1,\cdots,a_n}(\mathcal{B}, c) - \frac{1}{M}\right)^2 = \frac{M-1}{M|\mathcal{B}|}$$

holds, where $P_{a_1,\cdots,a_n}(\mathcal{B}, c)$ denotes the probability that $\sum_{i=1}^n a_i x_i \equiv c \pmod{M}$ for a random $(x_1, \cdots, x_n)$ drawn uniformly from $\mathcal{B}$, i.e.:

$$P_{a_1,\cdots,a_n}(\mathcal{B}, c) = \frac{1}{|\mathcal{B}|} \left| \left\{ (x_1, \cdots, x_n) \in \mathcal{B} \text{ such that } \sum_{i=1}^{n} a_i x_i \equiv c \pmod{M} \right\} \right|.$$

This implies the immediate corollaries:

Corollary 1. *For any real $\lambda > 0$, the fraction of n-tuples $(a_1, \cdots, a_n) \in \mathbb{Z}_M^n$ for which there exists a $c \in \mathbb{Z}_M$ that satisfies $|P_{a_1,\cdots,a_n}(\mathcal{B}, c) - 1/M| \geq \lambda/M$ is at most:*

$$\frac{M^2}{\lambda^2 \, |\mathcal{B}|}.$$

Corollary 2. *For any reals $\lambda > 0$ and $1 > \mu > 0$, the fraction of n-tuples $(a_1, \cdots, a_n) \in \mathbb{Z}_M^n$ for which there exist at least $\mu \, M$ values $c \in \mathbb{Z}_M$ that satisfy $|P_{a_1,\cdots,a_n}(\mathcal{B}, c) - 1/M| \geq \lambda/M$ is at most:*

$$\frac{M}{\lambda^2 \, \mu \, |\mathcal{B}|}.$$

These two corollaries are used when $|\mathcal{B}|$ is larger than M. We also need two more corollaries, one for small values of $|\mathcal{B}|$ and one for $|\mathcal{B}| \approx M$:

Corollary 3. *For any reals $1 > \mu > 0$, if $m > 1$ denotes $M/|\mathcal{B}|$, the fraction of n-tuples $(a_1, \cdots, a_n) \in \mathbb{Z}_M^n$ such that less than $\mu \, |\mathcal{B}|$ values $c \in \mathbb{Z}_M$ have $P_{a_1,\cdots,a_n}(\mathcal{B}, c) \neq 0$ is at most:*

$$\frac{\mu}{(1 - \mu)m}$$

Corollary 4. *For any reals $\lambda > 0$, the fraction of n-tuples $(a_1, \cdots, a_n) \in \mathbb{Z}_M^n$ that satisfy:*

$$\sum_{c \in \mathbb{Z}_M} P_{a_1,\cdots,a_n}(\mathcal{B}, c)^2 \geq \frac{M + |\mathcal{B}|}{\lambda M |\mathcal{B}|}$$

is at most λ.

3 The Algorithm of Schroeppel and Shamir

The algorithm of Schroeppel and Shamir was introduced in [21,22]. It allows to solve a generic integer knapsack problem on n-elements in time $\tilde{O}(2^{n/2})$ using a memory of size $\tilde{O}(2^{n/4})$. It improves on the birthday algorithm of Horowitz and Sahni [10] that can be applied on such a knapsack. We first recall this basic birthday algorithm, which is based on the rewriting of a knapsack solution as an equality:

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

where all the ϵs are 0 or 1. Thus, to solve the knapsack problem, we construct the set $\mathcal{S}^{(1)}$ containing all possible sums of the first $\lfloor n/2 \rfloor$ elements and $\mathcal{S}^{(2)}$ be the set obtained by subtracting from the target S any of the possible sums of the last $\lceil n/2 \rceil$ elements. Searching for collisions between the two sets, we discover all the solutions of the knapsack problem. This can be done in time and memory $\tilde{O}(2^{n/2})$ by fully computing the two sets, sorting them and looking up for collisions. In [21,22], Schroeppel and Shamir show that, in order to find these collisions, it is not necessary to store the full sets $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$. Instead, they generate them on the fly using priority queues (based either on heaps or a Adelson-Velsky and Landis trees), requiring memory $\tilde{O}(2^{n/4})$.

Algorithm 1. Schroeppel-Shamir algorithm

```
Require: Knapsack element $a_1, \ldots, a_n$. Knapsack sum $S$
  Let $q_1 = \lfloor n/4 \rfloor$, $q_2 = \lfloor n/2 \rfloor$, $q_3 = \lfloor 3\,n/4 \rfloor$
  Create $\mathcal{S}_L^{(1)}(\sigma)$ and $\mathcal{S}_L^{(1)}(\epsilon)$: list of all $\sum_{i=1}^{q_1} \epsilon_i\, a_i$ and list of $\epsilon_{1 \cdots q_1}$ (in the same order)
  Create $\mathcal{S}_R^{(1)}(\sigma)$ and $\mathcal{S}_R^{(1)}(\epsilon)$: list of all $\sum_{i=q_1+1}^{q_2} \epsilon_i\, a_i$ and list of $\epsilon_{q_1+1 \cdots q_2}$
  Create $\mathcal{S}_L^{(2)}(\sigma)$ and $\mathcal{S}_L^{(2)}(\epsilon)$: list of all $\sum_{i=q_2+1}^{q_3} \epsilon_i\, a_i$ and list of $\epsilon_{q_2+1 \cdots q_3}$
  Create $\mathcal{S}_R^{(2)}(\sigma)$ and $\mathcal{S}_R^{(2)}(\epsilon)$: list of all $\sum_{i=q_3+1}^{n} \epsilon_i\, a_i$ and list of $\epsilon_{q_3+1 \cdots n}$
  Call 4-way merge Algorithm 2 or 3 on $(\mathcal{S}_L^{(1)}(\sigma), \mathcal{S}_R^{(1)}(\sigma), \mathcal{S}_L^{(2)}(\sigma), \mathcal{S}_R^{(2)}(\sigma))$, $n$ and $S$.
  Store returned set in Sol
  for each $(i, j, k, l)$ in Sol do
    Concatenate $\mathcal{S}_L^{(1)}(\epsilon)[i]$, $\mathcal{S}_R^{(1)}(\epsilon)[j]$, $\mathcal{S}_L^{(2)}(\epsilon)[k]$ and $\mathcal{S}_R^{(2)}(\epsilon)[l]$ into $\epsilon$
    Output: "$\epsilon$ is a solution"
  end for
```

Algorithm 2. Original 4-Way merge routine

```
Require: Four input lists $(\mathcal{S}_L^{(1)}, \mathcal{S}_R^{(1)}, \mathcal{S}_L^{(2)}, \mathcal{S}_R^{(2)})$, knapsack size $n$, target sum $\mathcal{T}$
  Let $S_L^{(1)}$, $S_R^{(1)}$, $S_L^{(2)}$ and $S_R^{(2)}$ be the sizes of the corresponding arrays.
  Create priority queues $Q_1$ and $Q_2$
  Sort $\mathcal{S}_R^{(1)}$ and $\mathcal{S}_R^{(2)}$ in increasing order. Keep track of positions in InitPos$_1$ and InitPos$_2$
  for $i$ from 0 to $S_L^{(1)}$ do
    Insert $(i, 0)$ in $Q_1$ with priority $\mathcal{S}_L^{(1)}[i] + \mathcal{S}_R^{(1)}[0]$.
  end for
  for $i$ from 0 to $S_L^{(2)}$ do
    Insert $(i, S_R^{(2)} - 1)$ in $Q_2$ with priority $\mathcal{T} - \mathcal{S}_L^{(2)}[i] - \mathcal{S}_R^{(2)}[S_R^{(2)} - 1]$.
  end for
  Create empty list Sol
  while $Q_1$ and $Q_2$ are not empty do
    Peek at value $q_1$ of lowest priority element in $Q_1$.
    Peek at value $q_2$ of lowest priority element in $Q_2$.
    if $q_1 \le q_2$ then
      Get $(i, j)$ from $Q_1$
      if $j \ne S_R^{(1)} - 1$ then
        Insert $(i, j+1)$ in $Q_1$ with priority $\mathcal{S}_L^{(1)}[i] + \mathcal{S}_R^{(1)}[j+1]$.
      end if
    end if
    if $q_1 \ge q_2$ then
      Get $(k, l)$ from $Q_2$
      if $l \ne 0$ then
        Insert $(k, l-1)$ in $Q_2$ with priority $\mathcal{T} - \mathcal{S}_L^{(2)}[k] - \mathcal{S}_R^{(2)}[l-1]$.
      end if
    end if
    if $q_1 = q_2$ then
      Add $(i, \mathrm{InitPos}_1[j], k, \mathrm{InitPos}_2[l])$ to Sol
    end if
  end while
  Return list of solutions Sol
```

More precisely, let us define $q_1 = \lfloor n/4 \rfloor$, $q_2 = \lfloor n/2 \rfloor$, $q_3 = \lfloor 3\,n/4 \rfloor$. We introduce four sets $\mathcal{S}_L^{(1)}$, $\mathcal{S}_R^{(1)}$, $\mathcal{S}_L^{(2)}$ and $\mathcal{S}_R^{(2)}$ of size $O(2^{n/4})$ defined as follows:

- $\mathcal{S}_L^{(1)}$ is the set of pairs $(\sum_{i=1}^{q_1} \epsilon_i\, a_i, \epsilon_{1 \cdots q_1})$ with $\epsilon_{1 \cdots q_1} \in \{0,1\}^{q_1}$;
- $\mathcal{S}_R^{(1)}$ is the set of $(\sum_{i=q_1+1}^{q_2} \epsilon_i\, a_i, \epsilon_{q_1+1 \cdots q_2})$ with $\epsilon_{q_1+1 \cdots q_2} \in \{0,1\}^{q_2-q_1}$;
- $\mathcal{S}_L^{(2)}$ is the set of $(\sum_{i=q_2+1}^{q_3} \epsilon_i\, a_i, \epsilon_{q_2+1 \cdots q_3})$ with $\epsilon_{q_2+1 \cdots q_3} \in \{0,1\}^{q_3-q_2}$;
- $\mathcal{S}_R^{(2)}$ is the set of $(\sum_{i=q_3+1}^{n} \epsilon_i\, a_i, \epsilon_{q_3+1 \cdots n})$ with $\epsilon_{q_3+1 \cdots n} \in \{0,1\}^{n-q_3}$.

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 $S = \sigma_L^{(1)} + \sigma_R^{(1)} + \sigma_L^{(2)} + \sigma_R^{(2)}$. We call this a *4-way merge problem.*

The algorithm of Schroeppel and Shamir is described in Algorithm 1, using their original 4-way merge Algorithm 2 as a subroutine. Note that, in Algorithm 1, we describe each set $\mathcal{S}_X^{(i)}$ as two lists $\mathcal{S}_X^{(i)}(\sigma)$ and $\mathcal{S}_X^{(i)}(\epsilon)$ stored in the same order.

3.1 A Variation on the Schroeppel and Shamir Algorithm

In practice, the need for priority queues of large size makes the algorithm of Schroeppel and Shamir harder to implement and to optimize. Indeed, using large priority queues either introduces an extra factor in the memory usage or unfriendly cache behavior. As a consequence, we would like to avoid priority queues altogether. In order to do this, we present a variation on their algorithm, inspired by an algorithm presented in [3] that solves the problem of finding 4 elements from 4 distinct lists with bitwise sum equal to 0. Note that, from a theoretical point of view, our variation is not as good as the original algorithm of Schroeppel and Shamir, because for some exceptional knapsacks, it requires more memory.

The idea is to choose a modulus M near $2^{(1/4-\varepsilon)\,n}$ and to remark that the 4-way merge condition implies $\sigma_L^{(1)} + \sigma_R^{(1)} \equiv S - \sigma_L^{(2)} - \sigma_R^{(2)} \pmod{M}$. As a consequence, for any solution of the knapsack, there exists a value σ_M, such that:

$$\sigma_M = (\sigma_L^{(1)} + \sigma_R^{(1)}) \bmod M = (S - \sigma_L^{(2)} - \sigma_R^{(2)}) \bmod M.$$

Since, we cannot guess the correct value of σ_M, we simply loop over all possible values. This gives a new 4-way merge Algorithm 3, which can be used as a replacement for the original subroutine in Algorithm 1.

Informally, for each test value of σ_M, Algorithm 3 constructs the set of all sums $\sigma_L^{(1)} + \sigma_R^{(1)}$ congruent to σ_M modulo M. This is done by sorting $\mathcal{S}_R^{(1)}$ by 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 $\mathcal{S}_R^{(1)}$. Using this method, we construct the set $\mathcal{S}^{(1)}$ of the birthday paradox algorithm as a disjoint union of smaller sets $\mathcal{S}^{(1)}(\sigma_M)$, which are created one at a time within the loop on σ_M in Algorithm 2. Similarly, we implicitly construct $\mathcal{S}^{(2)}$ as a disjoint union of $\mathcal{S}^{(2)}(\sigma_M)$, but do not store it, instead searching for matching values in $\mathcal{S}^{(1)}(\sigma_M)$.

Algorithm 3. Modular 4-Way merge routine

```
Require: Four input lists $(\mathcal{S}_L^{(1)}, \mathcal{S}_R^{(1)}, \mathcal{S}_L^{(2)}, \mathcal{S}_R^{(2)})$, size $n$, target sum $\mathcal{T}$
Require: Memory margin parameter: $\varepsilon$
  Let $M$ be a random modulus in $[2^{(1/4-\varepsilon)\,n}, 2 \cdot 2^{(1/4-\varepsilon)\,n}]$
  Create list $\mathcal{S}_R^{(1)}(M)$ containing pairs $(\mathcal{S}_R^{(1)}[i] \bmod M, i)$ where $i$ indexes all of $\mathcal{S}_R^{(1)}$
  Create list $\mathcal{S}_R^{(2)}(M)$ containing pairs $(\mathcal{S}_R^{(2)}[i] \bmod M, i)$ where $i$ indexes all of $\mathcal{S}_R^{(2)}$
  Sort $\mathcal{S}_R^{(1)}(M)$ and $\mathcal{S}_R^{(2)}(M)$ by values of the left member of each pair
  Create empty list $Sol$
  for $\sigma_M$ from 0 to $M-1$ do
    Empty the list $\mathcal{S}^{(1)}$ (or create the list if $\sigma_M = 0$)
    for $i$ from 1 to size of $\mathcal{S}_L^{(1)}$ do
      Let $\sigma_L^{(1)} = \mathcal{S}_L^{(1)}[i]$ and $\sigma_t = (\sigma_M - \sigma_L^{(1)}) \bmod M$
      Binary search first occurrence of $\sigma_t$ in $\mathcal{S}_R^{(1)}(M)$
      for each consecutive $(\sigma_t, j)$ in $\mathcal{S}_R^{(1)}(M)$ do
        Add $(\sigma_L^{(1)} + \mathcal{S}_R^{(1)}[j]), (i,j))$ to $\mathcal{S}^{(1)}$
      end for
    end for
    Sort list $\mathcal{S}^{(1)}$ by values of the left member of each pair
    for $k$ from 1 to size of $\mathcal{S}_L^{(2)}$ do
      Let $\sigma_L^{(2)} = \mathcal{S}_L^{(2)}[k]$ and $\sigma_t = (\mathcal{T} - \sigma_M - \sigma_L^{(2)}) \bmod M$
      Binary search first occurrence of $\sigma_t$ in $\mathcal{S}_R^{(2)}$
      for each consecutive $(\sigma_t, l)$ in $\mathcal{S}_R^{(2)}(M)$ do
        Let $\mathcal{T}' = \mathcal{T} - \sigma_L^{(1)} - \mathcal{S}_R^{(2)}[l]$
        Binary search first occurrence of $\mathcal{T}'$ in $\mathcal{S}^{(1)}$
        for each consecutive $(T, (i,j))$ in $\mathcal{S}^{(1)}$ do
          Add $(i,j,k,l)$ to $Sol$
        end for
      end for
    end for
  end for
  Return list of solutions $Sol$
```

Algorithm 4. Our simple algorithm (Section 4.2)

```
Require: Knapsack elements $a_1, \ldots, a_n$. Knapsack sum $S$. Parameter $\beta$
  Let $M$ be a random prime close to $2^{\beta\,n}$
  Let $R_1$, $R_2$ and $R_3$ be random values modulo $M$.
  Solve the 1/8-unbalanced knapsack modulo $M$ with elements $a$ and target $R_1$.
  Solve the 1/8-unbalanced modular knapsack with target $R_2$.
  Solve the 1/8-unbalanced modular knapsack with target $R_3$.
  Solve the 1/8-unbalanced modular knapsack with target $S - R_1 - R_2 - R_3 \bmod M$.
  Create the 4 sets of non-modular sums corresponding to the above solutions.
  Do a 4-way merge (with early abort and consistency checks) on these 4 sets.
  Rewrite the obtained solution as a knapsack solution.
```

Complexity analysis. If we ignore the innermost loop that writes down the solution set *Sol*, the running time of the execution of the loop iteration corresponding to σ_M is $\tilde{O}(\mathcal{S}^{(1)}(\sigma_M) + \mathcal{S}^{(2)}(\sigma_M))$, with a polynomial factor in n that comes from sorting and searching. Summing over all iterations of the loop, we have a total running time of $\tilde{O}(\mathcal{S}^{(1)} + \mathcal{S}^{(2)}) = \tilde{O}(2^{n/2})$, unless *Sol* has a size larger than $\tilde{O}(2^{n/2})$.

Where memory is concerned, storing $\mathcal{S}_L^{(1)}$, $\mathcal{S}_R^{(1)}$, $\mathcal{S}_L^{(2)}$ and $\mathcal{S}_R^{(2)}$ costs $O(2^{n/4})$. However, the memory required to store the partitioned representation of $\mathcal{S}^{(1)}$ is $\max_{\sigma_M} \mathcal{S}^{(1)}(\sigma_M)$. Note that we cannot guarantee that this maximum remains small. A simple counterexample occurs when all a_i values (in the first half) are multiples of M. Indeed, in that case we find that $\mathcal{S}^{(1)}(0)$ has size $2^{n/2}$. In general, we do not expect such a bad behavior, more precisely, we have:

Theorem 2. *For any real $\varepsilon > 0$ and modulus M close to $2^{(1/4-\varepsilon)\,n}$, for a fraction at least $1 - 2^{-4\varepsilon\,n}$ of knapsacks with density $D < 4$ given by n-tuples $(a_1, \cdots, a_n)$ and target value T, Algorithm 1 using as 4-way merge routine Algorithm 3 finds all of the N_{Sol} solutions of the knapsack in time $\tilde{O}(\max(2^{n/2}, N_{Sol}))$ using memory $\tilde{O}(\max(2^{(1/4+\varepsilon)\,n}, N_{Sol}))$.*

Proof. The time analysis is given above. The bound on the memory use for almost all knapsacks comes from applying Corollary 1 with $\lambda = 1/2$ twice on the left and right-hand side subknapsacks on $n/2$ elements, using $\mathcal{B} = \{0,1\}^{n/2}$. We need to use the fact that a random knapsack taken uniformly at random with n/D-bit numbers is close to a random knapsack modulo M, when $D < 4$.

A high bit version. Instead of using modular values to partition the sets $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$, another option is to look at the value of the $\lceil n/4 \rceil$ higher bits. Depending on the precise context, this option might be more practical than the modular version. In the implementation presented in Section 5, we make use of both versions.

Early abort with multiple solutions. When the number of solutions N_{Sol} is large, and we wish to find a single solution, we can use an early abort strategy. Heuristically , assuming that the σ_M values corresponding to the many solutions are well-distributed modulo M, this reduces the heuristic expected running time to $\tilde{O}(\max(2^{n/4}, 2^{n/2}/N_{Sol}))$.

3.2 Application to Unbalanced Knapsacks

The basic birthday algorithm, the algorithm of Schroeppel–Shamir and our variation can also, with some care, be applied to α-unbalanced knapsacks. In this case, if we let:

$$\mathcal{C}_\alpha = \left(\alpha^{-\alpha} \cdot (1-\alpha)^{\alpha-1}\right),$$

the time complexity is $\tilde{O}(\mathcal{C}_\alpha^{n/2})$ and the memory complexity is $\tilde{O}(\mathcal{C}_\alpha^{n/2})$ for the basic birthday algorithm, $\tilde{O}(\mathcal{C}_\alpha^{n/4})$ for the algorithm of Schroeppel and Shamir and $\tilde{O}(\mathcal{C}_\alpha^{(1/4+\varepsilon)\,n})$ for our variation.

Adapting to the unbalanced case. Letting $\ell = \lfloor \alpha n \rfloor$, if we assume that the solution of the knapsack has $\lfloor \ell/2 \rfloor$ elements coming from the first half, then the algorithms are easily adapted. With the basic birthday method, the only difference with the balanced case is that $\mathcal{S}^{(1)}$ now contains all sums of exactly $\lfloor \ell/2 \rfloor$ elements among the $\lfloor n/2 \rfloor$ first elements and $\mathcal{S}^{(2)}$ contains all sums of $\lceil \ell/2 \rceil$ among the last $\lceil n/2 \rceil$ elements. This restriction is important, because allowing more elements on either side makes the sets $\mathcal{S}^{(1)}$ or $\mathcal{S}^{(2)}$ too large and prevents us from reaching the expected complexity bound. With balanced knapsacks, this is not an issue because $\binom{n}{\lfloor n/2 \rfloor}$ and 2^n are within polynomial factors of each other.

However, nothing *a priori* guarantees that the solution satisfies the above assumption. If it does, we say, following [24], that we have a splitting family. When n is even, to obtain such a splitting family, we can use a method attributed to Coppersmith in [24]. The idea is to run the algorithm n times on n knapsacks whose target sums are all equal to S and whose elements are rotated copies of $a_1, \ldots, a_n$. Namely, the elements of the i-th knapsack are $a_j^{(i)} = a_{(i+j) \bmod n}$. To prove that this works, it suffices to show that a sliding windows of $n/2$ consecutive elements intersects the solution S in exactly $\lfloor \ell/2 \rfloor$ points at least once, see [24] for details. When n is odd, we instead attempt to solve the two knapsacks on $n-1$ elements a_1 to a_{n-1} and targets S and $S - a_n$, thus going back to the even case. Alternatively, it is also possible to use a randomized approach also due to Coppersmith and described in [24]. In fact, it suffices to randomize the order of the a_i for each new trial and take the first and second halves. Thanks to Stirling's formulae, this, on average, only requires $O(\sqrt{n})$ trials.

For applying the algorithm of Schroeppel–Shamir or our variation to unbalanced knapsacks, we need to assume that the number of elements in each of the four quarters is known in advance and is either equal to $\lfloor \ell/4 \rfloor$ or to $\lceil \ell/4 \rceil$. Assuming that n is a multiple of 4, this can be achieved in a deterministic way by first using a sliding windows to guarantee that the two halves contains $\lfloor \ell/2 \rfloor$ or to $\lceil \ell/2 \rceil$ elements, then, inside of each half, we use another sliding window to balance the number of elements within the corresponding quarter. At most, we need to try $n^3/4$ configurations. When n is not a multiple of 4, we first guess the value of ϵ in $(n \bmod 4)$ positions and we are back to a knapsack with a number of elements equal to a multiple of 4. It is also possible to use a randomized approach, with an expected number of trials $O(n^{3/2})$.

4 The New Algorithms

4.1 Basic Principle

In this section, we want to solve a generic knapsack problem on n-elements. We start from the basic knapsack equation:

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

As explained in Section 2, by taking the complementary knapsack if required, we may assume that $\ell = \sum_{i=1}^{n} \epsilon_i \geq \lceil n/2 \rceil$.

We define the set $\mathcal{S}_{\lceil \ell/2 \rceil}$ as the set of all partial sums of $\lfloor \ell/2 \rfloor$ or $\lceil \ell/2 \rceil$ knapsack elements. Clearly, there exists pairs (σ_1, σ_2) of elements of $\mathcal{S}_{\lceil \ell/2 \rceil}$ such that $S = \sigma_1 + \sigma_2$. In fact, there exist many such pairs, corresponding to all the possible decompositions of the set of ℓ elements appearing in S into two subsets of size $\leq \lceil \ell/2 \rceil$. The number $\mathcal{N}_n$ of such decompositions is given either by the binomial $\binom{\ell}{\ell/2}$ for even ℓ or by $2\binom{\ell}{(\ell-1)/2}$ for odd ℓ.

The basic idea that underlies all algorithms presented in this paper is to focus on a small part on $\mathcal{S}_{\lceil \ell/2 \rceil}$, in order to discover one of these many solutions. We start by choosing a prime integer M near $\mathcal{N}_n$ and a random element R modulo M. Heuristically, we find that 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 $\mathcal{S}_{\lceil \ell/2 \rceil}$ containing elements respectively congruent to R and $S - R$ modulo M. Using the asymptotic estimates of binomials, we find that the expected size of each of these subsets is:

$$\frac{\binom{n}{\lceil \ell/2 \rceil}}{M} \approx \frac{\binom{n}{\lceil \ell/2 \rceil}}{\binom{\ell}{\lceil \ell/2 \rceil}} = \tilde{O}(2^{0.3113\,n}).$$

The exponent 0.3113 is obtained by approximating the binomial in the worst case where $\ell \approx n/2$. Once these two subsets, respectively denoted by $\mathcal{S}^{(1)}_{\lceil \ell/2 \rceil}$ and $\mathcal{S}^{(2)}_{\lceil \ell/2 \rceil}$ are constructed, we need to find a collision between σ_1 and $S - \sigma_2$, with σ_1 in $\mathcal{S}^{(1)}_{\lceil \ell/2 \rceil}$ and σ_2 in $\mathcal{S}^{(2)}_{\lceil \ell/2 \rceil}$. Clearly, using a classical sort and match method, this can be done in time $\tilde{O}(2^{0.3113\,n})$. As a consequence, assuming that we can construct the sets $\mathcal{S}^{(1)}_{\lceil \ell/2 \rceil}$ and $\mathcal{S}^{(2)}_{\lceil \ell/2 \rceil}$ quickly enough ,we can hope to construct an algorithm with overall complexity $\tilde{O}(2^{0.3113\,n})$ for solving generic knapsacks,. 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. The above idea can directly be applied to unbalanced knapsacks with $\ell = \alpha n$ elements in the decomposition of S. This expected size of the subsets of $\mathcal{S}_{\lceil \ell/2 \rceil}$ can now be approximated by:

$$\frac{\binom{n}{\lceil \ell/2 \rceil}}{\binom{\ell}{\lceil \ell/2 \rceil}} = \tilde{O}\left(\left(\frac{2}{\alpha^{\alpha/2} \cdot (2-\alpha)^{(2-\alpha)/2}}\right)^n \cdot 2^{-\alpha n}\right).$$

Interestingly, when $\alpha < 1/2$ we obtain a smaller bound by considering the complementary knapsack. As a consequence, in order to preserve the usual convention $\alpha \leq 1/2$, it is useful to substitute α by $1 - \alpha$, we obtain the bound:

$$\tilde{O}\left(\left((1-\alpha)^{(\alpha-1)/2} \cdot (1+\alpha)^{-(1+\alpha)/2}\right)^n \cdot 2^{\alpha n}\right).$$

The curve of the logarithm in base 2 of this bound is included in Figure 1.

4.2 Simple Algorithm

We first present a reasonably simple algorithm, which can achieve several trade-offs between time and memory. For simplicity, we assume that $\ell = \sum_{i=1}^{n} \epsilon_i = \lfloor n/2 \rfloor$. Should this not be the case, it would suffice to run the algorithm (possibly in the unbalanced version described below) for all values of $\ell \le \lfloor n/2 \rfloor$. In such a sequence of executions, the instance with $\ell = \lfloor n/2 \rfloor$ dominates the running time and the total run time remains within the same bound.

In our simple algorithm, instead of considering decompositions of S into two sub-sums as in the previous section, we now consider decompositions into four parts and write:

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

where each σ_i belongs to the set $\mathcal{S}_{\lceil \ell/4 \rceil}$ of all partial sums of either $\lfloor \ell/4 \rfloor$ or $\lceil \ell/4 \rceil$ knapsack elements. The exact number $\mathcal{N}$ of such decompositions varies depending on the value of ℓ modulo 4, for example:

$$\mathcal{N} = \binom{\ell}{\ell/4, \ell/4, \ell/4, \ell/4} \quad \text{when } \ell \equiv 0 \pmod 4.$$

However, in any case, thanks to Stirling's formula, we find that $\mathcal{N} = \tilde{O}(2^n)$.

We now choose an integer M near $2^{\beta n}$ (with $1/4 < \beta < 1/3$) and three random elements R_1, R_2 and R_3 modulo M. We then search for a decomposition that satisfies the constraints $\sigma_1 \equiv R_1 \pmod M$, $\sigma_2 \equiv R_2 \pmod M$, $\sigma_3 \equiv R_3 \pmod M$ and $\sigma_4 \equiv S - R_1 - R_2 - R_3 \pmod M$. Clearly, the fourth condition is a consequence of the other three and we heuristically expect $\mathcal{N} M^{-3}$ solutions that satisfy the extra constraints. To make this heuristic expectation precise enough we need the following generalization to Corollary 2:

Corollary 5. *When* $\log_2(M) > (3\log_2(3)/16)\, n \approx 0.2972\, n$*, for any reals* $\lambda > 0$ *and* $1 > \mu > 0$*, the fraction of* n*-tuples* $(a_1, \cdots, a_n) \in \mathbb{Z}_M^n$ *for which there exist at least* μM^3 *values* $(c_1, c_2, c_3) \in \mathbb{Z}_M$ *that satisfy* $|P_{a_1,\cdots,a_n}(\mathcal{B}, c_1, c_2, c_3) - 1/M^3| \ge \lambda/M^3$ *is at most:*

$$\frac{2M^3}{\lambda^2 \, \mu \, |\mathcal{B}|},$$

where $\mathcal{B}$ *is the set of decomposition of a given solution as* $(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)})$ *and* $P_{a_1,\cdots,a_n}(\mathcal{B}, c_1, c_2, c_3)$ *denotes the probability of the event:*

$$\sum_{i=1}^{n} a_i x_i^{(1)} \equiv c_1 \quad \textit{and} \quad \sum_{i=1}^{n} a_i x_i^{(2)} \equiv c_2 \quad \textit{and} \quad \sum_{i=1}^{n} a_i x_i^{(3)} \equiv c_3 \pmod M.$$

Proof. Refer to long version of this paper [12].

Heuristically, we also expect the corollary to hold as long as $\beta > 1/4$.

Once we have random values R_1, R_2 and R_3 that match a decomposition of the solution, we can find the solution as follows: we start by constructing the four subsets of $\mathcal{S}_{\lceil \ell/4 \rceil}$ 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 $\mathcal{S}^{(1)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(2)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(3)}_{\lceil \ell/4 \rceil}$ and $\mathcal{S}^{(4)}_{\lceil \ell/4 \rceil}$. Once this is done, we search for a knapsack solution by doing a 4-way merge of these sets. This strategy is outlined as Algorithm 4.

Constructing the subsets. To construct each of the subsets $\mathcal{S}^{(1)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(2)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(3)}_{\lceil \ell/4 \rceil}$ and $\mathcal{S}^{(4)}_{\lceil \ell/4 \rceil}$, we use the algorithm of Schroeppel and Shamir. Note that, since the solution we are searching is a sum of $\lfloor \ell/4 \rfloor$ or $\lceil \ell/4 \rceil$ elements, we need to use the algorithm in the unbalanced case, with $\alpha = 1/8$. Depending on the value of β, the set of solutions may be quite large. Indeed, the expected number of solutions is $\binom{n}{\lceil n/8 \rceil} \cdot 2^{-\beta n} = \tilde{O}(2^{(0.5436-\beta)\, n})$. Since this is bigger than the size of the subsets $\mathcal{S}^{(i)}_{\lceil \ell/4 \rceil}$, the memory complexity of Algorithm 1 is $\tilde{O}(2^{(0.5436-\beta)\, n})$, while its time complexity is $\tilde{O}(\max(2^{(0.5436-\beta)\, n}, 2^{0.272\, n}))$. For the theoretical analysis, we assume here that we are using the original 4-way merge algorithm of Schroeppel and Shamir whose complexity is always guaranteed.

Of course, since we are solving modular knapsack instances, we first need to transform the problems into (polynomially many instances of) integer knapsacks as explained in Section 2. In any case, note that the time and memory requirements of this stage are dominated by the complexity of the next stage.

Recovering the desired solution. Once the subsets $\mathcal{S}^{(1)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(2)}_{\lceil \ell/4 \rceil}$, $\mathcal{S}^{(3)}_{\lceil \ell/4 \rceil}$ and $\mathcal{S}^{(4)}_{\lceil \ell/4 \rceil}$ are constructed, it suffices to perform a 4-way merge of these sets using a slightly modified version of the modular[2] 4-way merge Algorithm 3. For this 4-way merge, we use a modulus M' coprime to M. We choose M' close to $\binom{n}{\lceil \ell/4 \rceil} 2^{-\beta n} \approx 2^{(0.5436-\beta) n}$.

The changes to Algorithm 3 are the following:

1. Rename the modulus as M'
2. Replace the "for" loop on the $\sigma_{M'}$ value, by a loop where each new value of $\sigma_{M'}$ is randomly selected.
3. At each merge, i.e. insertion in $\mathcal{S}^{(1)}$, *Sol* or (implicit) $\mathcal{S}^{(2)}$, add a consistency check to make sure that the corresponding subset sums do not overlap. If consistency check fails, skip the insertion.
4. Add an early abort criteria: stop the algorithm at the first insertion in *Sol*.

At the end of the algorithm, the consistent solution $\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4 = S$ present in *Sol* can be translated into a solution of the knapsack problem.

Complexity analysis (sketch of proof). We have already seen that the time complexity of the subset construction phase is $\tilde{O}(\max(2^{(0.5436-\beta) n}, 2^{0.272\, n}))$ using memory $\tilde{O}(2^{(0.5436-\beta)\, n})$. To analyze the complexity of the recovery stage, we need to know the size of the intermediate set of sums $\mathcal{S}^1(\sigma_{M'})$. Note that

[2] Here, we cannot use the original 4-way merge, because we do not know how to analyze its complexity when early abort is used.

this set contains all choices of $\lceil \ell/2 \rceil$ elements among n that can be written as a sum $\sigma = \sigma_1 + \sigma_2$ satisfying a modular constraints, i.e., $\sigma_1 \equiv \sigma_{M'} \pmod{M'}$. By construction, we also have $\sigma_1 \equiv R_1 + R_2 \pmod{M}$.

Using the same techniques, we can also show that there exists a constant τ such that at least $\tau \min(2^{(1-3\beta)\,n}, 2^{(1/2-\beta)\,n})$ decompositions of the original solutions in two parts with $\sigma_1 \equiv R_1 + R_2 \pmod{M}$ are obtained. Let $\mathcal{B}$ denotes this set of accessible decompositions and look at the corresponding sums modulo $M' > |\mathcal{B}|$. Applying Corollary 3 with $\mu = 1/2$, we find that, for all but an exponentially small fraction of n-tuples $(a_1, \cdots, a_n)$, at least $|\mathcal{B}|/2$ different sums modulo M'. As a consequence, since the $\sigma_{M'}$ values are taken at random, the 4-way merge requires an expected number of iterations $M'/(2|\mathcal{B}|)$. Moreover, after $n\,M'/(2|\mathcal{B}|)$ iterations there is an overwhelming probability to find at least one such decomposition. Thus, the early abort occurs after $\tilde{O}(M'/(2|\mathcal{B}|))$ iterations.

It remains to analyze the time complexity of each iteration of the loop. It is dominated by the number of merged pairs that need to be tested for consistency. For any value of $\sigma_{M'}$ the number of pairs is the sum over c of the number of elements congruent to c modulo M' in the first list by the number of elements congruent to $\sigma_{M'} - c$ modulo M' in the second list. This is a scalar product of two vectors on M' elements. It is smaller than the product of the norms of the two vectors. We can bound the squared norm using Corollary 4, with $\lambda = 2^{-\varepsilon n}$. We find that for an exponentially small fraction λ of n-tuples, the number of pairs tested for consistency per iteration is $\tilde{O}(2^{\varepsilon n} M')$. Multiplying by the number of iterations, we find a total time $\tilde{O}(2^{\varepsilon n} {M'}^2/|\mathcal{B}|) = \tilde{O}(2^{(0.0872+\beta)\,n})$ when ε is small enough.

We should also state that the number of quadruples tested for consistency is $\tilde{O}(2^{0.3113\,n})$. Putting everything together, when $1/3 > \beta > 1/4$, we summarize the overall running time of the algorithm as $\tilde{O}(\max(2^{0.3113\,n}, 2^{(0.0872+\beta)n}))$ using $\tilde{O}(2^{(0.5436-\beta)n})$ units of memory. We recall that, when $\beta \leq 3\log_2(3)/16$ the analysis is only heuristic.

Some possible time-memory trade-offs. We now instantiate this simple algorithm by choosing values for β. A first option is to minimize the required amount of memory, this is achieved by taking β arbitrarily close to $1/3$ and yields a running time $\tilde{O}(2^{0.421\,n})$, using $\tilde{O}(2^{0.211\,n})$ memory units. A second option is to look at the smallest value of β for which we can prove the algorithm, i.e., $\beta \approx 0.2972$, we have running time $\tilde{O}(2^{0.385\,n})$, using $\tilde{O}(2^{0.247\,n})$ memory units. A third heuristic option is to require the same amount of memory as in Schroeppel–Shamir, i.e. $\tilde{O}(2^{n/4})$, this occurs for $\beta \approx 0.2936$ and corresponds to a running time $\tilde{O}(2^{0.381\,n})$. Finally, we can minimize the running time by taking β close to $1/4$ and obtain an algorithm with time complexity $\tilde{O}(2^{0.338\,n})$ and memory complexity $\tilde{O}(2^{0.294\,n})$.

For the choices $\beta < 1/4$, the time complexity becomes $\tilde{O}(2^{(0.5872-\beta)n})$ and increases again.

Complexity for unbalanced knapsacks. As in Section 3.1, this algorithm can be extended to α-unbalanced knapsacks, with $\alpha \leq 1/2$. Writing the time complexity as $\tilde{O}(2^{\mathcal{C}_\alpha n})$ and the memory complexity as $\tilde{O}(2^{\mathcal{D}_\alpha n})$, we have:

$$\mathcal{C}_\alpha = 2\log_2\left(\frac{4}{\alpha^{\alpha/4}\cdot(4-\alpha)^{(4-\alpha)/4}}\right) - 2\alpha + 2\beta\alpha \quad \text{and}$$
$$\mathcal{D}_\alpha = \log_2\left(\frac{4}{\alpha^{\alpha/4}\cdot(4-\alpha)^{(4-\alpha)/4}}\right) - 2\beta\alpha.$$

As in the balanced case, the parameter β determines the chosen time-memory trade-off.

Knapsacks with multiple solutions. Note that nothing prevents the above algorithm from finding a large fraction of the solutions for knapsacks with many solutions. However, in that case, we need to take some additional precautions. We need to change the early abort strategy and to remove any duplicate representation of a given solution. We should remember that, if the number of solutions becomes too large it can dominate time and memory complexities.

For an application that would require all the solutions of the knapsack, it is also 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}. Heuristically, to balance this, we increase the running time by a factor of $\log(N_{Sol})$.

4.3 A Better Heuristic Algorithm

Despite the fact that the algorithm from Section 4.2 outperforms the method of Schroeppel and Shamir, it does not achieve the complexity expected from Section 4.1. Admittedly, with the choice of β that optimizes speed, it comes reasonably close. However, in this case, it requires more memory than we would expect. In order to further reduce the complexity, we propose a heuristic algorithm that more closely follows the basic idea from Section 4.1. More precisely, we need to write $S = \sigma_1 + \sigma_2$ and constrain σ_1 enough to lower the number of expected solutions close to 1. Once again, we assume, for simplicity, that n is even and that we are considering a 1/2-unbalanced knapsack.

We choose a modulus M close to $2^{\gamma n}$ with $\gamma \geq 1/2$ and thus need to consider on average $2^{(\gamma-1/2)n}$ different random values for σ_1 modulo M. For each of these $2^{(\gamma-1/2)n}$ random values, denoted by 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 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. After transforming the problem into integer knapsack problems, we simply use the algorithm from Section 4.2 in the 1/4-unbalanced case. This can be done using time $\tilde{O}(2^{0.2996\,n})$ and memory $\tilde{O}(2^{0.2123\,n})$, assuming that the number of solutions is no bigger than that. Since the number of expected solutions is $\binom{n}{n/4}/M = \tilde{O}(2^{(0.8113-\gamma)n})$, we choose γ between 0.5117 and 0.5990, in order to balance the number of solutions returned by the subroutine with some compromise between its time or memory. Here is a table that shows some achievable trade-offs:

γ	Time exponent	Memory exponent	Comment
0.5117	0.3113	0.2996	Lowest time
0.5177	0.3173	0.2936	Same memory as Algorithm 4
0.5375	0.3372	0.2737	Same time as Algorithm 4
0.5613	0.3609	1/4	Same memory as Schroeppel-Shamir
0.5990	0.3986	0.2123	Lowest memory

The behavior of this algorithm for α-unbalanced knapsacks is shown on Figure 1.

Complexity using recursion. Finally, we can use this heuristic approach recursively to slightly reduce the memory requirements. In fact, one level of recursion is enough. To solve an α-unbalanced knapsack, we cut it in two halves and solve the resulting $(\alpha/2)$-unbalanced knapsacks using the above heuristic method. Thanks to the faster runtime of the subroutine, we can use a different choice for γ and obtain the lowest runtime with less memory. More precisely, the memory use for α-unbalanced knapsacks is now equal to the running time of the heuristic algorithm on $(\alpha/2)$-unbalanced knapsacks. As a consequence, we can solve 1/2-unbalanced knapsacks in time $\tilde{O}(2^{0.3113\,n})$ using $\tilde{O}(2^{0.2936\,n})$ units of memory.

5 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.

Variation of Schroeppel–Shamir algorithm. Concerning the implementation of Schroeppel–Shamir 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 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 Schroeppel–Shamir presented in Section 3.1 with a prime modulus $M = 2\,704\,157$. Testing the full space of solutions requires $120 \times 37 = 4\,400$ days on a single Intel core 2 duo at 2.66 Ghz. 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 to enumerate the search space. The memory requirements are either 300 Mbytes of memory with initial lists of size $\binom{24}{12}$ and 1.8 Gbytes with initial lists of size 2^{24}.

Of course, the algorithm may succeed before finishing the full enumeration.

Our simple algorithm. As with the Schroeppel–Shamir 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 less than an hour on the same computer. When such a decomposition is not known in advance, we need about 120 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 $M = 1\,253\,839$.
– For the final merging of the four obtained lists, we use the modulus 2 493 709 and apply consistency checks and early abort.

The memory requirement are approximately 2.6 Gbytes of memory.

Our better heuristic algorithm. In our implementation of the algorithm, the small lists that occur at the innermost level are so small that we replaced Schroeppel–Shamir algorithm there by a basic birthday paradox method. Thus, we no 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 instead of 120 times. The parameters we use are:

– For the higher level modulus, we choose $M = 4\,194\,319 \cdot 58\,711 \cdot 613$.
– The innermost birthday paradox method is done modulo 613.
– Assembling the two half-knapsacks is performed modulo $58\,711 \cdot 613$.

In practice, using such composite moduli saves time and memory. With the above parameters, our implementation uses about 1.7 Gbytes and runs in approximately 1.5 hours given a correct decomposition[3] into two halves. Without such a decomposition, we need less than 10 hours to find a solution.

6 Possible Extensions and Applications

The algorithmic techniques presented in this paper can be applied to more than ordinary knapsacks. We already mentioned modular knapsacks in Section 2, we now describe a few more:

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

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

[3] More precisely, we found 30 copies of the solution in 157 585 seconds on a single core.

where δ is small, i.e. belongs to the range $[-B, B]$ for a specified bound B. As the modular knapsack problem, this 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 $S' - \lceil n/2 \rceil, \ldots, S' + \lceil n/2 \rceil$.

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 used 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 apply similar methods. However, we obtain different bounds, since the number of different representations of a given solution is no longer the same. For simplicity of presentation, we assume that n is a multiple of 3 and that the solution contains $n/3$ values of each type. A simple birthday approach works by searching for a collision between two sums of $n/3$ knapsack elements. It is equal to:

$$\binom{n}{n/3} \approx \tilde{O}(2^{0.9183\,n}).$$

Note that this is higher than the expected $3^{n/2}$. A slightly more complex approach splits the knapsack in two halves and search for a collision between a left and right sum, each containing one third each of of 0, 1 and -1. This yields the expected complexity $3^{n/2}$. Using our ideas and taking a collision between two half-sums each containing two-thirds of 0s and one sixth of each of 1 and -1 of the n elements, we find a complexity $\tilde{O}(2^{0.585\,n})$ to find one of the $2^{2n/3}$ possible decompositions.

Single solution out of many. When there are many possible solutions to a knapsack problem, we may wish to combine our idea with the generalized birthday algorithm of [25] and find one of the many solutions even faster. However, this approach is difficult to analyze in general.

Combination of the above and possible applications. In fact, it is even possible to address combinations 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 [11] 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 [8], also needs to be studied with our new algorithm in mind. Another possible application would be the SWIFFT hash function [17].

Note that, in all cases, our algorithms never affect asymptotic security of a cryptographic scheme, indeed, an algorithm with complexity $2^{0.3113n}$ remains exponential. However, depending on the initial designers hypothesis, recommended practical parameters may need to be increased. For the special case of NTRU, it can be seen that in [9] that the estimates are conservative enough not to be affected by our algorithms.

7 Conclusion, Open Problems

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 31-year old algorithm of Schroeppel and Shamir and gives a positive answer to the question posed in the Open Problem Garden [6] about knapsack problems: "Is there an algorithm that runs in time $2^{n/3}$?". Many interesting related problems are still open:

– Find a fast deterministic algorithm to solve the knapsack problem. In particular, such an algorithm could show that a given knapsack does not have a solution.
– Devise a fast Las Vegas algorithm, i.e., a randomized algorithm that can prove that a given knapsack has no solution.
– Improve our algorithms by using a full recursive approach.
– Reduce the memory requirements. Surprisingly, general cycle finding techniques do not seem to apply in this case and do not yield a constant memory algorithm with time $\tilde{O}(2^{n/2})$.

Acknowledgements. We would like to thank Igor Shparlinski for useful discussions about exponential sums and distributions of knapsack outputs. The first author also appreciates time spent thinking about this problem at NTRU Cryptosystems, Inc.

References

1. Ajtai, M.: The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In: 30th ACM STOC, Dallas, Texas, USA, May 23–26, pp. 10–19. ACM Press, New York (1998)
2. Camion, P., Patarin, J.: The Knapsack hash function proposed at Crypto'89 can be broken. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 39–53. Springer, Heidelberg (1991)
3. Chose, P., Joux, A., Mitton, M.: Fast correlation attacks: An algorithmic point of view. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 209–221. Springer, Heidelberg (2002)

4. Coster, M.J., Joux, A., LaMacchia, B.A., Odlyzko, A.M., Schnorr, C.-P., Stern, J.: Improved low-density subset sum algorithms. Computational Complexity 2, 111–128 (1992)
5. Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
6. Open problem garden, `http://garden.irmacs.sfu.ca`
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)
8. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC, Bethesda, MD, USA, May 2009, pp. 169–178. ACM Press, New York (2009)
9. Hirschorn, P.S., Hoffstein, J., Howgrave-Graham, N., Whyte, W.: Choosing NTRU-Encrypt parameters in light of combined lattice reduction and MITM approaches. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 437–455. Springer, Heidelberg (2009)
10. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. J. Assoc. Comp. Mach. 21(2), 277–292 (1974)
11. Howgrave-Graham, N.: A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 150–169. Springer, Heidelberg (2007)
12. Howgrave-Graham, N., Joux, A.: New generic algorithms for hard knapsacks, `eprint.iacr.org` or `www.joux.biz/publications/Knapsacks.pdf`
13. Impagliazzo, R., Naor, M.: Efficient cryptographic schemes provably as secure as subset sum. Journal of Cryptology 9(4), 199–216 (1996)
14. Joux, A., Granboulan, L.: A practical attack against knapsack based hash functions (extended abstract). In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 58–66. Springer, Heidelberg (1994)
15. Lagarias, J.C., Odlyzko, A.M.: Solving low-density subset sum problems. J. Assoc. Comp. Mach. 32(1), 229–246 (1985)
16. Lenstra, A.K., Lenstra Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. 261, 515–534 (1982)
17. Lyubashevsky, V., Micciancio, D., Peikert, C., Rosen, A.: Swifft: A modest proposal for fft hashing. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 54–72. Springer, Heidelberg (2008)
18. Merkle, R., Hellman, M.: Hiding information and signatures in trapdoor knapsacks. IEEE Trans. Information Theory 24(5), 525–530 (1978)
19. Nguyen, P.Q., Shparlinski, I.E., Stern, J.: Distribution of modular sums and the security of the server aided exponentiation. Progress in Computer Science and Applied Logic 20, 331–342 (2001); Final Proceedings of Cryptography and Computational Number Theory workshop, Singapore (1999)
20. Schnorr, C.-P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science 53, 201–224 (1987)
21. Schroeppel, R., Shamir, A.: A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. In: FOCS, pp. 328–336 (1979)
22. Schroeppel, R., Shamir, A.: 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)
23. Shamir, A.: A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology – CRYPTO 1982, Santa Barbara, CA, USA, pp. 279–288. Plenum Press, New York (1983)

24. Stinson, D.R.: Some baby-step giant-step algorithms for the low hamming weight discrete logarithm problem. Math. Comput. 71(237), 379–391 (2002)
25. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)

A Graph of Compared Complexities

In the following figure, we present the complexity of the algorithms discussed in the paper for α-unbalanced knapsacks.

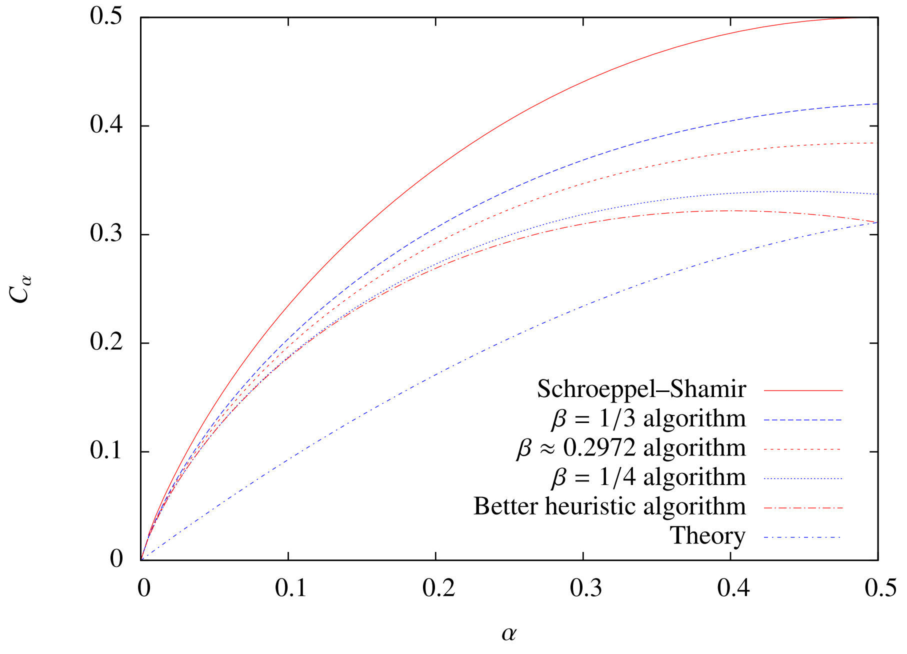

Fig. 1. Curves of time complexity exponent for varying balance factor