Improved Exponential-time Algorithms for Inhomogeneous-SIS

Shi Bai[1], Steven D. Galbraith[1], Liangze Li[2], and Daniel Sheffield[1]

[1] Department of Mathematics, University of Auckland, Auckland, New Zealand.
[2] School of Mathematical Sciences, Peking University, Beijing, China.

Abstract. The paper is about algorithms for the inhomogeneous short integer solution problem: Given $(\mathbf{A}, \mathbf{b})$ to find a short vector $\mathbf{s}$ such that $\mathbf{As} \equiv \mathbf{b} \pmod q$. We consider algorithms for this problem due to Camion and Patarin; Wagner; Schroeppel and Shamir; Howgrave-Graham and Joux; Becker, Coron and Joux. Our main results include: Applying the Hermite normal form (HNF) to get faster algorithms; A heuristic analysis of the HGJ and BCJ algorithms in the case of density greater than one; An improved cryptanalysis of the SWIFFT hash function.
Keywords: SIS, subset-sum

1 Introduction

The *subset-sum* problem (also called the "knapsack problem") is: Given positive integers $a_1, \ldots, a_m$ and an integer s, to compute a vector $\mathbf{x} = (x_1, \ldots, x_m) \in \{0,1\}^m$ if it exists, such that

$$s = \sum_{i=1}^{m} a_i x_i.$$

It is often convenient to write $\mathbf{a} = (a_1, \ldots, a_m)$ as a row and $\mathbf{x} = (x_1, \ldots, x_m)^T$ as a column so that $s = \mathbf{ax}$. The *modular subset-sum* problem is similar: Given a modulus q, integer vector $\mathbf{a}$ and integer s to find $\mathbf{x} \in \{0,1\}^m$, if it exists, such that $s \equiv \mathbf{ax} \pmod q$.

The vector version of this problem is called the *inhomogenous shortest integer solution problem* (ISIS): Given a modulus q, a small set $\mathcal{B} \subseteq \mathbb{Z}$ that contains 0 (e.g., $\mathcal{B} = \{0,1\}$ or $\{-1,0,1\}$), an $n \times m$ matrix $\mathbf{A}$ (where typically m is much bigger than n), and a column vector $\mathbf{s} \in \mathbb{Z}_q^n$ to find a column vector $\mathbf{x} \in \mathcal{B}^m$ (if it exists) such that

$$\mathbf{s} \equiv \mathbf{Ax} \pmod q. \tag{1}$$

If we want to be more precise we call this the $(m, n, q, \mathcal{B})$-ISIS problem. The original shortest integer solution problem (SIS) is the case $\mathbf{s} = 0$, in which case it is required to find a solution $\mathbf{x} \in \mathcal{B}^m$ such that $\mathbf{x} \neq 0$.

We unify the subset-sum and ISIS problems as the $(G, m, \mathcal{B})$-ISIS problem where G is an abelian group (written additively), m an integer and $\mathcal{B}$ a small subset of $\mathbb{Z}$ that contains 0. The three motivating examples for the group are $G = \mathbb{Z}$, $G = \mathbb{Z}_q$ and $G = \mathbb{Z}_q^n$. An instance of the problem is a pair $(\mathbf{A}, \mathbf{s})$ with $\mathbf{A} \in G^m$ and $\mathbf{s} \in G$, and a solution is any vector $\mathbf{x} \in \mathcal{B}^m$ (if one exists) such that $\mathbf{s} = \mathbf{Ax}$. We extend the notion of "density" for subset-sum problems, which captures the expected number of solutions $\mathbf{x}$ that exist for a random $(\mathbf{A}, \mathbf{s})$ pair.

These computational problems have applications in lattice-based cryptography. For example, inverting the SWIFFT hash function of Lyubashevsky, Micciancio, Peikert and Rosen [16] is solving $(1024, 64, 257, \{0,1\})$-ISIS. Since this function is a compression function (mapping 1024 bits to 512 bits) it corresponds to a very high density instance of ISIS. The security level of SWIFFT claimed

in [16] is "to find collisions takes time at least 2^{106} and requires almost as much space, and the known inversion attacks require about 2^{128} time and space".[3] Appendix B of an early version of [12] gives an improved collision attack, exploiting the birthday paradox, using lists of size 2^{96} (surprisingly this result is missing in the published version [11]). In fact these arguments are very approximate and do not give precise bounds on that the actual running time of these attacks. We remark that a stronger variant of this hash function has also been proposed [1], but we do not discuss it further in this paper.

It is known that one can try to solve both subset-sum and ISIS using lattice methods (for example, reducing to the closest vector problem or shortest vector problem in a certain lattice of dimension m or $m+1$). However, the focus in this paper is on algorithms based on time-memory tradeoffs. It is important to take into account both lattice algorithms and time-memory tradeoff algorithms when selecting parameters for lattice-based cryptosystems. For this reason, we assume that the set $\mathcal{B}$ is rather small (e.g., $\mathcal{B} = \{0, 1\}$ or $\{-1, 0, 1\}$). Some previous algorithms of this type for the subset-sum and ISIS problems are due to: Schroeppel and Shamir; Camion and Patarin; Wagner; Minder and Sinclair; Howgrave-Graham and Joux; Becker, Coron and Joux. The Camion-Patarin/Wagner/Minder-Sinclair (CPW) method is suitable for very high density instances (such as SWIFFT), while the other methods are more suitable for low density instances. We will recall the previous algorithms in Section 2.

1.1 Our contributions

Our first contribution is to give a general framework that unifies the subset-sum, modular subset-sum and ISIS problems. We show that the algorithms by Schroeppel and Shamir, Camion and Patarin, Wagner, Howgrave-Graham and Joux, Becker-Coron-Joux can be used to solve these generalised problems. The three main contributions of our paper are:

1. To develop variants of these algorithms for the *approximate-ISIS* problem, which itself arises naturally when one takes the Hermite normal form of an ISIS instance. This problem also arises as the binary-LWE problem. This is done in Section 4.
2. To study the Howgrave-Graham and Joux (HGJ) and Becker-Coron-Joux (BCJ) methods in the case of instances of density greater than one. We give in Figure 1 of Section 3 a comparison of the HGJ, BCJ and CPW algorithms as the density grows.
3. To give improved cryptanalysis [4] of the SWIFFT hash function [16]. We reduce the collision attack time from around 2^{113} to around 2^{104} bit operations (a speed-up by a factor ≈ 500). We also reduce inverting time by a factor of ≈ 1000.

We focus on the general problem (with an arbitrary matrix $\mathbf{A}$, rather than problems coming from NTRU or Ring-LWE that have a structured matrix). The binary-LWE problem [17] (with both the "secret" and "errors" chosen to be binary vectors) is a case of the approximate-ISIS problem. Hence our algorithms can also be applied to this problem. Note that binary-LWE is not usually a high density problem.

[3] We remark that generic hash function collision algorithms such as parallel collision search would require at least 2^{256} bit operations. Hence we do not consider such algorithms further in this paper.

[4] We remark that in [5], the authors claimed that finding pseudo-collisions for SWIFFT is comparable to breaking a 68-bit symmetric cipher, however the pseudo-collision is not useful to find real collisions for SWIFFT, since they reduce to the sublattices of dimension 206 in which the real collisions for SWIFFT almost do not exist.

1.2 Related literature

There is an extensive literature on the approximate subset-sum problem over $\mathbb{Z}$ (to find $\mathbf{x}$ such that $\mathbf{s} \approx \mathbf{ax}$) including polynomial-time algorithms (see Section 35.5 of [7]). These algorithms exploit properties of the usual ordering on $\mathbb{Z}$ and do not seem to be applicable to ISIS. Indeed, such algorithms cannot be directly applied to the modular subset-sum problem either, though the modular subset-sum problem can be lifted to polynomially many instances of the subset-sum problem over $\mathbb{Z}$ and then the approximate subset-sum algorithms can be applied. Hence, even though the algorithms considered in our paper can be applied to the subset-sum and modular subset-sum problems, our main interest is in the ISIS problem.

2 Algorithms to solve subset-sum/ISIS

2.1 A general framework

We propose the following general framework for discussing the algorithms of Camion and Patarin, Wagner, Minder and Sinclair, Howgrave-Graham and Joux, Becker, Coron and Joux. Previously they were always discussed in special cases.

We define the $(G, m, \mathcal{B})$-ISIS problem where G is an abelian group, m an integer and $\mathcal{B}$ a small subset of $\mathbb{Z}$ that contains 0. Our three main examples for the group are $G = \mathbb{Z}$, $G = \mathbb{Z}_q$ and $G = \mathbb{Z}_q^n$. An instance of the problem is a pair $(\mathbf{A}, \mathbf{s})$ with $\mathbf{A} \in G^m$ and $\mathbf{s} \in G$, and a solution is any vector $\mathbf{x} \in \mathcal{B}^m$ (if one exists) such that $\mathbf{s} = \mathbf{Ax}$.

The *weight* of a solution $\mathbf{x}$ is defined to be $\mathrm{wt}(\mathbf{x}) = \#\{i : 1 \leq i \leq m, a_i \neq 0\}$. One can consider the weight-ω $(G, m, \mathcal{B})$-ISIS problem, where the input is $(\mathbf{A}, \mathbf{s})$ and where one is asked to compute a solution $\mathbf{x} \in \mathcal{B}^m$ such that $\mathbf{s} = \mathbf{Ax}$ in G and $\mathrm{wt}(\mathbf{x}) = \omega$.

All the algorithms work by reducing to simpler problems of higher density. In our general framework we express this by taking quotients. Let H be a subgroup of G and write G/H for the quotient. Since the map $G \to G/H$ is a group homomorphism, an instance $\mathbf{s} = \mathbf{Ax}$ in G reduces to an instance $\mathbf{s} \equiv \mathbf{Ax} \pmod{H}$ in G/H. The density increases since the number of possible targets $\mathbf{s} \pmod{H}$ is reduced while the number of inputs $\mathbf{x}$ remains the same. In practice we will employ this idea in the following ways: when $G = \mathbb{Z}$ then $H = M\mathbb{Z}$ and $G/H = \mathbb{Z}_M$; when $G = \mathbb{Z}_q$ and $M \mid q$ then $H = M\mathbb{Z}_q$ and $G/H \cong \mathbb{Z}_M$; when $G = \mathbb{Z}_q^n$ then $H = \{(0, \ldots, 0, g_{\ell+1}, \ldots, g_n)^T : g_i \in \mathbb{Z}_q\} \cong \mathbb{Z}_q^{n-\ell}$ so that $G/H \cong \mathbb{Z}_q^\ell$.

Now we define density, which is a standard concept in the subset-sum problem but is a little harder to define for ISIS. Consider the $(G, m, \mathcal{B})$-ISIS problem where G is now finite. Let δ be the probability, over uniformly chosen elements $(\mathbf{A}, \mathbf{s})$ in $G^m \times G$, that there exists a solution $\mathbf{x} \in \mathcal{B}^m$ such that $\mathbf{s} = \mathbf{Ax}$. If δ is small then we say the $(G, m, \mathcal{B})$-ISIS problem has low density. If $\delta \approx 1$ then consider the average size (over uniform choices $(\mathbf{A}, \mathbf{s})$) of the set of solutions $\{\mathbf{x} \in \mathcal{B}^m : \mathbf{s} = \mathbf{Ax}\}$. If this set has average size close to 1 then we say we are in the "density 1" case. If this set is large on average then we are in the high density case. This informal notion is consistent with the standard notion of density for the subset-sum problem over $\mathbb{Z}$. (The standard notion can be formulated by defining the vector/matrix $\mathbf{A}$ to have entries in an interval $[0, B] \subset \mathbb{Z}$.)

High density instances can always be reduced to smaller dimensional instances having density one: Choose a suitable integer ℓ and set ℓ entries of $\mathbf{x}$ to be zero. Delete the corresponding columns from $\mathbf{A}$ to get an $n \times (m - \ell)$ matrix $\mathbf{A}'$ and let $\mathbf{x}'$ be the corresponding solution vector in $\mathbb{Z}^{m-\ell}$. Then solve the density one problem $\mathbf{A}'\mathbf{x}' \equiv \mathbf{s} \pmod{q}$. When evaluating algorithms for high density

ISIS we must always compare them against the best low-density algorithms when applied to the reduced problem.

2.2 Brief survey of previous methods

It is straightforward that one can solve the $(G, m, \{0,1\})$-ISIS problem in $\tilde{O}(2^{m/2})$ time and large storage using birthday methods. Schroeppel and Shamir [20] showed how to match this running time but use considerably less space. A simpler description of the Schroeppel-Shamir algorithm was given by Howgrave-Graham and Joux [11]. We briefly recall some details in Section 2.4.

The important paper of Howgrave-Graham and Joux [11] (HGJ) breaks the $\tilde{O}(2^{m/2})$ barrier, giving a heuristic algorithm to solve subset-sum in $\tilde{O}(2^{0.337m})$ operations, and with large storage (around $\tilde{O}(2^{0.256m})$). Note that [11] presents algorithms for the traditional subset-sum problem, but Section 6 of [11] mentions that the methods should be applicable to variants of the subset-sum problem including approximate subset-sum, vector versions of subset-sum (i.e., ISIS), and different coefficient sets (e.g., $x_i \in \{-1, 0, 1\}$). Our paper thus addresses these predictions from [11]; we give the details in Section 2.7. Indeed, it is written in [11] that "It would be interesting to re-evaluate the security of SWIFFT with respect to our algorithm."

Becker, Coron and Joux [2] gave some improvements to the HGJ method (also restricted to the setting of subset-sum). We sketch the details in Section 2.8.

Camion and Patarin [6] gave an algorithm for solving high density subset-sum instances, and similar ideas were used by Wagner [22] for solving the "k-sum problem". Rather unfairly, these ideas are now called "Wagner's algorithm", but we will call it CPW. Minder and Sinclair [19] explained how to use these ideas a bit more effectively.

In 2004, Lyubashevsky noted that the CPW algorithm can be applied to solve high density subset-sum problems. Shallue [21] extended Lyubashevsky's work. It was noted in Lyubashevsky, Micciancio, Peikert and Rosen [16] that the CPW algorithm can be applied to solve ISIS in the high density case (inverting the SWIFFT hash function is a very high density case of ISIS).

All known algorithms are obtained by combining two basic operations (possibly recursively):

1. Compute lists of solutions to some constrained problem obtained by "splitting" the solution space (i.e., having a smaller set of possible $\mathbf{x}$) in a quotient group G/H. Splitting the solution space lowers the density, but working in the quotient group G/H compensates by raising the density again.
2. Merge two lists of solutions to give a new list of solutions in a larger quotient group G/H'.

The algorithms differ primarily in the way that splitting is done.

2.3 The merge algorithm

We consider one step of the merge algorithm[5].

We now introduce the notation to be used throughout. Let $\mathcal{X} \subseteq \mathcal{B}^m$ be a set of coefficients. We will always be working with a set of subgroups $\{H_i : 1 \leq i \leq t\}$ of G such that, for each pair $1 \leq i < j \leq t$ we have $\#(G/(H_i \cap H_j)) = \#(G/H_i) \cdot \#(G/H_j)$. We denote the size of G/H_i by M_i. Since our main interest is the case $G = \mathbb{Z}_q^n$ we will sometimes write M_i as q^{ℓ_i}. All algorithms

[5] The word "merge" is not really appropriate as we are not computing a union or intersection of lists, but forming sums $\mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1 \in L_1$ and $\mathbf{x}_2 \in L_2$. However, it is the word used by several previous authors.

involve splitting the set of coefficients $\mathcal{X} \subseteq \mathcal{X}_1 + \mathcal{X}_2 = \{\mathbf{x}_1 + \mathbf{x}_2 : \mathbf{x}_1 \in \mathcal{X}_1, \mathbf{x}_2 \in \mathcal{X}_2\}$ in some way (for example, by positions or by weight).

Let $H^\flat, H, H^\sharp$ be three subgroups of G. The desired output of the merge algorithm is a set of solutions to the problem $\mathbf{Ax} \equiv \mathbf{s} \pmod{H \cap H^\flat}$ for $\mathbf{x} \in \mathcal{X}$, together with information about $\mathbf{Ax} \pmod{H^\sharp}$ to be used in future calculations. The input is a pair of lists L_1 and L_2 that are "partial solutions" modulo $H^\flat$. In other words we are merging modulo H, lists of partial solutions modulo $H^\flat$, that are going to be used for a future computation modulo $H^\sharp$. The algorithm is given as Algorithm 1.

Algorithm 1 Basic merge algorithm

INPUT: $L_1 = \{(\mathbf{x}, \mathbf{Ax} \pmod{H}) : \mathbf{Ax} \equiv R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_1\}$,
$L_2 = \{(\mathbf{x}, \mathbf{Ax} \pmod{H}) : \mathbf{Ax} \equiv \mathbf{s} - R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_2\}$
OUTPUT: $L = \{(\mathbf{x}, \mathbf{Ax} \pmod{H^\sharp}) : \mathbf{Ax} \equiv \mathbf{s} \pmod{H \cap H^\flat}, \mathbf{x} \in \mathcal{X}\}$
1: Initialise $L = \{\}$
2: Sort L_2 with respect to the second coordinate
3: **for** $(\mathbf{x}_1, \mathbf{u}) \in L_1$ **do**
4: Compute $\mathbf{v} = \mathbf{s} - \mathbf{u} \pmod{H}$
5: **for** $(\mathbf{x}_2, \mathbf{v}) \in L_2$ **do**
6: **if** $\mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{X}$ **then**
7: Compute $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H^\sharp}$
8: Add $(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H^\sharp})$ to L

The running time of the algorithm depends on the cost of sorting L_2, which is $O(\#L_2 \log_2(\#L_2))$ i.e. $\tilde{O}(\#L_2)$. However, the time is often dominated by the total number of pairs $(\mathbf{x}_1, \mathbf{x}_2)$ considered in the algorithm, and this depends on how many values $\mathbf{u}$ there are in common between the two lists L_1 and L_2. Treating the function from $\mathcal{X}$ to G/H given by $\mathbf{x} \mapsto \mathbf{Ax} \pmod{H}$ as pseudorandom, the total number of $(\mathbf{x}_1, \mathbf{x}_2)$ pairs can be bounded by $\#L_1 \cdot \#L_2/M$, where $M = \#(G/H)$. Hence, the heuristic running time is $\tilde{O}(\max\{\#L_2, \#L_1\#L_2/M\})$. (Our analysis includes the correction by May and Meurer to the analysis in [11], as mentioned in Section 2.2 of [2].)

Another remark is that, in many cases, it is non-trivial to bound the size of the output list L. Instead, this can be bounded by $\#\mathcal{X}/\#(G/(H \cap H^\flat))$.

2.4 Schroeppel and Shamir algorithm

Schroeppel and Shamir [20] noted that by using 4 lists instead of 2 one could get an algorithm for subset-sum over $\mathbb{Z}$ with the same running time but with storage growing proportional to $(\#\mathcal{B})^{m/4}$. (Their presentation is more general than just subset-sum over $\mathbb{Z}$.)

Howgrave-Graham and Joux obtained this result in a much simpler way by using reduction modulo M and Algorithm 1. Our insight is to interpret reduction modulo M as working in a quotient group G/H. It immediately follows that the HGJ formulation of the Schroeppel-Shamir algorithm is applicable to the $(G, m, \mathcal{B})$-ISIS problem, giving an algorithm that requires time proportional to $(\#\mathcal{B})^{m/2}$ and space proportional to $(\#\mathcal{B})^{m/4}$. Since our goal is to discuss improved algorithms, we do not give the details here.

Dinur, Dunkelman, Keller and Shamir [9] have given improvements to the Schroeppel-Shamir algorithm, in the sense of getting a better time-memory curve. However, their methods always require time at least $(\#\mathcal{B})^{m/2}$. Since we are primarily concerned with reducing the average running time, we do not consider their results further.

2.5 Camion and Patarin/Wagner algorithm (CPW)

This algorithm is applicable for instances of very high density. It was first proposed by Camion and Patarin for subset-sum, and then by Wagner in the additive group $\mathbb{Z}_2^m$ (and some other settings). Section 3 of Micciancio and Regev [18] notes that the algorithm can be used to solve (I)SIS.

Let $k = 2^t$ be a small integer such that $k \mid m$. Let $H_1, \cdots, H_t$ be subgroups of the abelian group G such that

$$G \cong (G/H_1) \oplus \cdots \oplus (G/H_t). \tag{2}$$

Precisely we need that $G/(H_{i_1} \cap \cdots \cap H_{i_u}) \cong (G/H_{i_1}) \oplus \cdots \oplus (G/H_{i_u})$ and $H_1 \cap \cdots \cap H_t = \{0\}$. One can think of this as being like a "Chinese remainder theorem" for G: there is a one-to-one correspondence between G and the set of t-tuples $(g \pmod{H_1}, \ldots, g \pmod{H_t})$. We usually require that $\#(G/H_i)$ is roughly $(\#G)^{1/(t+1)}$ for $1 \leq i < t$ and $\#(G/H_t) \approx (\#G)^{2/(t+1)}$, although Minder and Sinclair [19] obtain improvements by relaxing these conditions.

For (I)SIS problem, we have $G = \mathbb{Z}_q^n$. Let $\ell \in \mathbb{N}$ be such that $\ell \approx n/(t+1)$. Then we choose the subgroup $H_1 = \{(0, \ldots, 0, g_{\ell+1}, \ldots, g_n)^T : g_i \in G\}$ such that $G/H_1 \cong \mathbb{Z}_q^\ell$ corresponds to the first ℓ positions of the vector. Similarly, G/H_2 corresponds to the next ℓ positions of the vector (so $H_2 = \{(g_1, \ldots, g_\ell, 0, \ldots, 0, g_{2\ell+1}, \ldots, g_n)^T\}$). Finally, G/H_t corresponds to the last $\approx 2\ell$ positions of the vector.

The "splitting" in the CPW approach is by positions. To be precise, let $u = m/k$ and define $\mathcal{X}_1 = \{(x_1, \ldots, x_u, 0, \ldots, 0) \in \mathcal{B}^m\}$ and

$$\mathcal{X}_j = \{(0, \ldots, 0, x_{(j-1)u+1}, \ldots, x_{ju}, 0, \ldots, 0) \in \mathcal{B}^m\}$$

for $2 \leq j \leq k$. The CPW algorithm works by first constructing $k = 2^t$ lists $L_j^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_j\}$ for $1 \leq j \leq k-1$ and $L_k^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x} - \mathbf{s}) : \mathbf{x} \in \mathcal{X}_k\}$. Each list consists of $\#\mathcal{X}_j = (\#\mathcal{B})^u$ elements and can be computed in $O((\#\mathcal{B})^u) = O((\#\mathcal{B})^{m/2^t})$ operations in G. (To optimise the running time one only computes $\mathbf{A}\mathbf{x} \pmod{H_1}$ at this stage.) The aim is to find vectors $\mathbf{x}_j \in L_j^{(0)}$ for $1 \leq j \leq k$ such that $\sum_j \mathbf{A}\mathbf{x}_j = 0$.

The algorithm proceeds by "merging" the lists pairwise. One computes new lists $L_1^{(1)}, \ldots, L_{k/2}^{(1)}$, where each $L_j^{(1)}$ contains pairs $(\mathbf{x}_1, \mathbf{x}_2) \in L_{2j-1}^{(0)} \times L_{2j}^{(0)}$ such that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \equiv 0 \pmod{H_1}$. In other words, the new lists contain elements $\mathbf{x}_1 + \mathbf{x}_2$ that are "correct" for the quotient G/H_1. The merge can be performed efficiently using Algorithm 1. The next step is to merge the lists $L_{2j-1}^{(1)}$ and $L_{2j}^{(1)}$ to get $L_j^{(2)}$ by ensuring the solutions are correct modulo H_2, and so on.

We refer to [6, 22, 18, 16, 19] for full details and heuristic analysis. The standard argument is that we want the lists $L^{(1)}, L^{(2)}, \ldots$ to all be roughly the same size, so that they are large enough for the following stages of the algorithm to succeed. It follows that we desire $(\#\mathcal{B})^{2m/k}/(\#G)^{1/(t+1)} \approx (\#\mathcal{B})^{m/k}$ and so $(\#G)^{1/(t+1)} \approx (\#\mathcal{B})^{m/k}$. In practice one takes $k = 2^t$ to be as large as possible subject to this constraint, and the size of k is governed by the density of the instance (higher density means larger k).

The main drawbacks of the CPW algorithm are: it requires very large storage (the time and memory complexity are approximately equal); it is not amenable to parallelisation; it can only be used for very high density instances. Some techniques to reduce storage and benefit from parallelism are given by Bernstein et al [3, 4]. Note that algorithm is completely deterministic, and so always gives the same solution set, but to obtain other solutions one can apply a random permutation

to the problem before running the algorithm. Another remark is that when the density is 1 (i.e., $(\#\mathcal{B})^m \approx q^n$) then we need to have $k = 1 + \log_2(k)$ and hence $k = 2$, and the CPW algorithm becomes the trivial "meet-in-middle" method.

Our general framework allows to consider the CPW algorithm for subset-sum and modular subset-sum. However, to have a decomposition as in equation (2) one needs the modulus in the modular subset-sum problem to have factors of a suitable size. Wagner's paper mentions an approach for modular subset-sum using sub-intervals instead of quotients (for further details see Lyubashevsky [15]). We also mention the work of Shallue [21], which gives a rigorous analysis of the CPW algorithm for the modular subset-sum problem.

Finally, we mention the work of Minder and Sinclair [19] that allows a finer balancing of parameters. This allows the CPW algorithm to be used for larger values of k than the density might predict. We sketch some details in Section 2.6.

2.6 Minder and Sinclair refinement of CPW

Minder and Sinclair [19] proposed the extended k-tree algorithm. In the previous section we divided the problem into k lists, and divided the group $G = \mathbb{Z}_q^n$ into t sections using subgroups $H_1, \ldots, H_t$ such that $\#(G/H_i) = q^\ell$ for $1 \leq i < t$ and $\#(G/H_t) \approx q^{2\ell}$. The new idea is to choose integers $\ell_1, \ell_2, \cdots, \ell_t$ and subgroups H_i so that $\#(G/H_i) = q^{\ell_i}$.

Denote by $L^{(i)}$ any of the lists at the i-th stage of the algorithm. Recall that we want to minimise $\max_{0 \leq i \leq t}(\#L^{(i)})$ and that we have

$$\#L^{(i)} \leq \#L^{(i-1)}\#L^{(i-1)}/q^{\ell_i}. \tag{3}$$

Since $\#L^{(0)} = (\#\mathcal{B})^{m/k}$ we can write $\#L^{(i)} = 2^{b_i}$ where $b_0 = (m/k)\log_2(\#\mathcal{B})$. It is usually the case that equation (3) is an equality, and hence get $b_i = 2b_{i-1} - \log_2(q)\ell_i$. So to minimize the time complexity the ℓ_i should be a solution of the following integer program:

$$\begin{aligned}
\text{minimize } & b_{max} = \max_{0 \leq i \leq t} b_i \\
\text{subject to } & 0 \leq b_i, \quad 0 \leq i \leq t \\
& b_0 = (m/k)\log_2(\#\mathcal{B}), \\
& b_i = 2b_{i-1} - \log_2(q)\ell_i, \\
& \ell_i \geq 0, \qquad 0 \leq i \leq t \\
& \sum_{i=1}^{t} \ell_i = n.
\end{aligned}$$

If $(\#\mathcal{B})^{m/k} \approx (\#G)^{1/(t+1)}$, the solution to the above integer program is $\ell_1 = \cdots = \ell_t \approx n/(1+t)$, and the extended k-tree algorithm in this case is the original CPW algorithm. When $(\#\mathcal{B})^{m/k} < (\#G)^{1/(t+1)}$ then it is not possible to use k lists in the CPW algorithm. However, the extended k-tree algorithm may still be useful with k lists, if one chooses appropriate values for ℓ_i.

2.7 The algorithm of Howgrave-Graham and Joux (HGJ)

We now present the HGJ algorithm, that can be applied even for density 1 instances of the $(G, m, \mathcal{B})$-ISIS problem and heuristically improves on the square-root time complexity of Schroeppel-Shamir.

For simplicity we focus on the case $\mathcal{B} = \{0, 1\}$. Section 6 of [11] notes that a possible extension is to develop the algorithm for "vectorial knapsack problems". Our formulation contains this predicted extension.

The first crucial idea of Howgrave-Graham and Joux [11] is to split the vector $\mathbf{x}$ by weight rather than by positions. The second crucial idea is to reduce to a simpler problem and then apply the algorithm recursively. The procedures in [11] use reduction modulo M, which we generalise as a map into a quotient group G/H. It follows that the HGJ algorithm can be applied to a more general class of problems.

Suppose we wish to solve $\mathbf{s} = \mathbf{A}\mathbf{x}$ in G where $\mathbf{x} \in \mathcal{B}^m$ has weight $\mathrm{wt}(\mathbf{x}) = \omega$. Write $\mathcal{X}$ for the set of weight ω vectors in $\mathcal{B}^m$, and write $\mathcal{X}_1, \mathcal{X}_2$ for the set of weight $\omega/2$ vectors in $\mathcal{B}^m$. Then there are $\binom{\omega}{\omega/2}$ ways to write $\mathbf{x}$ as $\mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1 \in \mathcal{X}_1, \mathbf{x}_2 \in \mathcal{X}_2$.

The procedure is to choose a suitable subgroup H so that there is a good chance that a randomly chosen element $R \in G/H$ can be written as $\mathbf{A}\mathbf{x}_1$ for one of the $\binom{\omega}{\omega/2}$ choices for $\mathbf{x}_1$. Then the procedure solves the two subset-sum instances in the group G/H (recursively) to generate lists of solutions

$$L_1 = \{\mathbf{x}_1 \in \mathcal{B}^m : \mathbf{A}\mathbf{x}_1 = R \pmod{H}, \mathrm{wt}(\mathbf{x}_1) = \omega/2\}$$

and

$$L_2 = \{\mathbf{x}_2 \in \mathcal{B}^m : \mathbf{A}\mathbf{x}_2 = \mathbf{s} - R \pmod{H}, \mathrm{wt}(\mathbf{x}_2) = \omega/2\}.$$

We actually store pairs of values $(\mathbf{x}_1, \mathbf{A}\mathbf{x}_1 \pmod{H'}) \in \mathcal{B}^m \times (G/H')$ for a suitably chosen subgroup H'. One then applies Algorithm 1 to merge the lists to get solutions $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{X}$ satisfying the equation in $G/(H \cap H')$. The paper [11] gives several solutions to this problem of merging lists, including a 4-list merge. But the main algorithm in [11] exploits Algorithm 1.

The subgroup H is chosen to trade-off the probability that a random value R corresponds to some splitting of the desired original solution $\mathbf{x}$ (this depends on the size of the quotient group G/H), while also ensuring that the lists L_1 and L_2 are not too large.

One inconvenience is that we may not exactly know the weight of the desired solution $\mathbf{x}$. If we can guess that the weight of $\mathbf{x}$ lies in $[\omega - 2\epsilon, \omega + 2\epsilon]$ then we can construct lists $\{\mathbf{x}_1 : \mathbf{A}\mathbf{x}_1 = R \pmod{H}, \mathrm{wt}(\mathbf{x}_1) \in [\omega/2 - \epsilon, \omega/2 + \epsilon]\}$. A similar idea can be used at the bottom level of the recursion, when we apply the Schroeppel-Shamir method and so need to split into vectors of half length and approximately half the weight.

The improvement in complexity for finding the solutions in L_1 and L_2 is due to the lowering of the weight from ω to $\omega/2$. This is why the process is amenable to recursive solution. At some point one terminates the recursion and solves the problem by a more elementary method (e.g. Schroeppel-Shamir).

One must pay attention to the relationship between the group G/H and the original group G. For example, when solving modular subset-sum in $G = \mathbb{Z}_q$ where q does not have factors of a suitable size then, as noted in [11], "we first need to transform the problems into (polynomially many instances of) integer knapsacks". For the case $G = \mathbb{Z}_q^n$ this should not be necessary.

Complexity analysis: The final algorithm is a careful combination of these procedures, performed recursively. We limit our discussion to recursion of 3 levels. In terms of the subgroups, the recursive nature of the algorithm requires a sequence of subgroups H_1, H_2, H_3 (of the same form as in Section 2.5) so that the quotient groups $G/(H_1 \cap H_2 \cap H_3)$, $G/(H_2 \cap H_3)$, G/H_3 become smaller and smaller. (The "top level" of the recursion turns an ISIS instance in G to two lower-weight ISIS

instances in $G' = G/(H_1 \cap H_2 \cap H_3)$; to solve these sub-instances using the same method we need to choose a quotient of G' by some proper subgroup $H_2 \cap H_3$, which is the same as taking a quotient of G by the subgroup $H_2 \cap H_3$.)

In [11], for subset-sum over $\mathbb{Z}$, this tower of subgroups is manifested by taking moduli M that divide one another ("For the higher level modulus, we choose $M = 4194319 \cdot 58711 \cdot 613$", meaning $H_3 = 613\mathbb{Z}, H_2 = 58711\mathbb{Z}$, $H_1 = 4194319\mathbb{Z}$, $H_2 \cap H_3 = (58711 \cdot 613)\mathbb{Z}$ and $H_1 \cap H_2 \cap H_3 = M\mathbb{Z}$). In the case of modular subset-sum in $\mathbb{Z}_q$ when q does not split appropriately one can lift to $\mathbb{Z}$ (giving a polynomial number of instances) and reduce each of them by a new composite modulus.

We do not reproduce all the analysis from [11], since it is superseded by the method of Becker et al. But the crucial aspect is that the success of the algorithm depends on the probability that there is a splitting $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ of the solution into equal weight terms such that $\mathbf{A}\mathbf{x}_1 = R \pmod{H}$. This depends on the number $\binom{\omega}{\omega/2}(\#\mathcal{B} - 1)^{\omega/2}$ of splittings of the weight ω vector $\mathbf{x}$ and on the size $M = \#(G/H)$ of the quotient group. Overall, the heuristic running time for the HGJ method (as stated in Section 2.2 of [2]) is $\tilde{O}(2^{0.337m})$.

2.8 The algorithm of Becker, Coron and Joux

Becker, Coron and Joux [2] present an improved version of the HGJ algorithm (again, their paper is in the context of subset-sum, but easily generalises to our setting). The idea is to allow larger coefficient sets. Precisely, suppose $\mathcal{B} = \{0, 1\}$ and let $\mathcal{X} \subset \mathcal{B}^n$ be the set of weight ω vectors. The HGJ idea is to split $\mathcal{X}$ by taking $\mathcal{X}_1 = \mathcal{X}_2$ to be the set of weight $\omega/2$ vectors in $\mathcal{B}^m$. Becker et al suggest to take $\mathcal{X}_1 = \mathcal{X}_2$ to be the set of vectors in $\mathbb{Z}^m$ having $\omega/2 + \alpha m$ entries equal to $+1$ and αm entries equal to -1, and the remaining entries equal to 0. This essentially increases the density of the sub-problems, and leads to a better choice of parameters. The organisation of the algorithm, and its analysis, are the same as HGJ. The HGJ algorithm is simply the case $\alpha = 0$ of the BCJ algorithm.

We briefly sketch the heuristic analysis from [2] for the case of 3 levels of recursion, $\mathcal{B} = \{0, 1\}$, balanced solution of weight $m/2$, and instances of density 1 (so that $2^m \approx q^n$). Let

$$\mathcal{X}_{a,b} = \{\mathbf{x} \in \{-1, 0, 1\}^m : \#\{i : x_i = 1\} = am, \#\{i : x_i = -1\} = bm\}.$$

A good approximation to $\#\mathcal{X}_{a,b}$ is $2^{mH(a,b)}$ where $H(x, y) = -x \log_2(x) - y \log_2(y) - (1 - x - y) \log_2(1 - x - y)$.

Fix $\alpha = 0.0267, \beta = 0.0168$ and $\gamma = 0.0029$ and also integers ℓ_1, ℓ_2, ℓ_3 such that $q^{\ell_1} \approx 2^{0.267m}, q^{\ell_2} \approx 2^{0.291m}$ and $q^{\ell_3} \approx 2^{0.241m}$. Choose subgroups H_1, H_2, H_3 such that $\#(G/H_i) = q^{\ell_i}$.

Theorem 1. *(Becker-Coron-Joux) With notation as above, and assuming heuristics about the pseudorandomness of $\boldsymbol{Ax}$, the BCJ algorithm runs in time $\tilde{O}(2^{0.291m})$.*

Proof. (Sketch) The first level of recursion splits $\mathcal{X} = \mathcal{B}^m$ into $\mathcal{X}_1 + \mathcal{X}_2$ where $\mathcal{X}_1 = \mathcal{X}_2 = \mathcal{X}_{1/4+\alpha,\alpha}$. We compute two lists $L_1^{(1)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_1, \mathbf{A}\mathbf{x} \equiv R_1 \pmod{H_1 \cap H_2 \cap H_3}\}$ and $L_2^{(1)}$, which is the same except $\mathbf{A}\mathbf{x} \equiv \mathbf{s} - R_1 \pmod{H_1 \cap H_2 \cap H_3}$. The expected size of the lists is $2^{H(1/4+\alpha,\alpha)m}/q^{\ell_1+\ell_2+\ell_3} = 2^{0.217m}$ and merging requires $\tilde{O}((2^{0.217m})^2/q^{n-\ell_1-\ell_2-\ell_3}) = \tilde{O}(2^{(2 \cdot 0.217 - 0.201)m}) = \tilde{O}(2^{0.233m})$ time.

The second level of recursion computes each of $L_1^{(1)}$ and $L_2^{(1)}$, by splitting into further lists. For example, $L_1^{(1)}$ is split into $L_1^{(2)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}, \mathbf{A}\mathbf{x} \equiv R_2 \pmod{H_2 \cap H_3}\}$ and

$L_2^{(2)}$ is similar except the congruence is $\mathbf{Ax} \equiv R_1 - R_2 \pmod{H_2 \cap H_3}$. Again, the size of lists is approximately $2^{H(1/8+\alpha/2+\beta,\alpha/2+\beta)m}/q^{\ell_2+\ell_3} = 2^{0.278m}$ and the cost to merge is $\tilde{O}(2^{(2\cdot 0.278-0.267)m}) = \tilde{O}(2^{0.289m})$.

The final level of recursion computes each $L_j^{(2)}$ by splitting into two lists corresponding to coefficient sets $\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}$. The expected size of the lists is

$$2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m}/q^{\ell_3} \approx 2^{0.291m}$$

and they can be computed efficiently using the Shroeppel-Shamir algorithm in time

$$\tilde{O}(\sqrt{2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m}}) = \tilde{O}(2^{0.266m}).$$

Merging the lists takes $\tilde{O}(2^{2*0.291}/q^{\ell_2}) = \tilde{O}(2^{0.291m})$ time.

The above theorem does not address the probability that the algorithm succeeds to output a solution to the problem. The discussion of this issue is complex and takes more than 3 pages (Section 3.4) of [2]. We give a rough "back-of-envelope" calculation that gives some confidence.

Suppose there is a unique solution $\mathbf{x} \in \{0,1\}^m$ of weight $m/2$ to the ISIS instance. Consider the first step of the recursion. For the whole algorithm to succeed, it is necessary that there is a splitting $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ of the solution so that $\mathbf{x}_1 \in L_1^{(1)}$ and $\mathbf{x}_2 \in L_2^{(1)}$. We split $\mathbf{x}$ so that the $m/2$ ones are equally distributed across $\mathbf{x}_1$ and $\mathbf{x}_2$, and the $m/2$ zeroes are sometimes expanded as $(-1,+1)$ or $(+1,-1)$ pairs. Hence, the number of ways to split $\mathbf{x}$ in this way is

$$\mathcal{N}_1 = \binom{m/2}{m/4}\binom{m/2}{\alpha m}\binom{(1/2-\alpha)m}{\alpha m} = \binom{m/2}{m/4}\binom{m/2}{\alpha m, \alpha m, (1/2-2\alpha)m}.$$

For randomly chosen $R_1 \in G/(H_1 \cap H_2 \cap H_3)$, there is a good chance to be a valid splitting if $\mathcal{N}_1 \approx q^{\ell_1+\ell_2+\ell_3}$. Indeed, the expected number of valid splittings should be roughly $\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3}$.

For the second stage we assume that we already made a good choice in the first stage, and indeed that we have $\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3}$ possible values for $\mathbf{x}_1$. The number of ways to further split $\mathbf{x}_1$ is $\mathcal{N}_2 = \binom{(1/4+\alpha)m}{(1/8+\alpha/2)m}\binom{\alpha m}{\alpha m/2}\binom{(3/4-2\alpha)m}{\beta m,\beta m,(3/4-2\alpha-2\beta)m}$. Hence, we require $(\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2 > 1$.

In the final stage (again assuming a good splitting in the second stage), the number of ways to split is $\mathcal{N}_3 = \binom{(1/8+\alpha/2+\beta)m}{(1/16+\alpha/4+\beta/2)m} \cdot \binom{(\beta+\alpha/2)m}{(\beta/2+\alpha/4)m} \cdot \binom{(7/8-\alpha-2\beta)m}{\gamma m,\gamma m,(7/8-\alpha-2\beta-2\gamma)m}$, which we require to be $\approx q^{\ell_3}$. Thus, choosing $\#G/H_3$ close to $\mathcal{N}_3$, $\#G/(H_2 \cap H_3)$ close to $\mathcal{N}_2$ and $\#G/(H_1 \cap H_2 \cap H_3)$ close to $\mathcal{N}_1$ then it is believed the success probability of the algorithm is a constant. This argument is supported in Section 3.4 of [2] by theoretical discussions and numerical experiments. To conclude, for the algorithm to have a good chance to succeed we require

$$\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3} > 1, \quad (\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2 > 1, \quad (\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2(\mathcal{N}_3/q^{\ell_3})^4 > 1.$$

2.9 Summary

Despite the large literature on the topic, summarised above, one sees there are only two basic ideas that are used by all algorithms:

- *Reduce modulo subgroups to create higher density instances.* Since the new instances have higher density one now has the option to perform methods that only find some of the possible solutions.

– *Splitting solutions.* Splitting can be done by length (i.e., positions) or by weight. Either way, one reduces to two "simpler" problems that can be solved recursively and then "merges" the solutions back to solutions to the original problem.

The main difference between the methods is that CPW requires large density to begin with, in which case splitting by positions is fine. Whereas HGJ/BCJ can be applied when the original instance has low density, in which case it is necessary to use splitting by weight in order to be able to ignore some potential solutions.

3 Analysis of HGJ/BCJ in high density

The CPW algorithm clearly likes high density problems. However, the analysis of the HGJ and BCJ algorithms in [11, 2] is in the case of finding a specific solution (and so is relevant to the case of density at most 1). It is intuitively clear that when the density is higher (and so there is more than one possible solution), and when we only want a single solution to the problem, then the success probability of the algorithm should increase. In this section we explain that the parameters in the HGJ and CPW algorithms can be improved when one is solving instances of density > 1. This was anticipated in [12]: "further improvements can be obtained if, in addition, we seek one solution among many". We give a very approximate heuristic analysis of this situation.

Let the number of the solutions to the original subset-sum problems be $\mathcal{N}_{sol} \geq 1$. We consider at most t levels of recursion. The subgroups $H_1, H_2, \cdots, H_t$ are chosen to trade-off the probability of a successful split at each stage and also to ensure the size of the lists to be merged at each stage is not too large. Using the same notation as Section 2.8, write $\mathcal{N}_1, \mathcal{N}_2, \cdots, \mathcal{N}_t$ for the number of ways to split a single valid solution at each level of the recursion.

The standard approach is to choose the subgroups $H_1, H_2, \cdots, H_t$ such that $\#G/(H_i \cap H_{i+1} \cap \cdots \cap H_t) = q^{\ell_i + \ell_{i+1} + \cdots + \ell_t} \approx \mathcal{N}_i$ for all $1 \leq i \leq t$. The success probability is then justified by requiring

$$\frac{\mathcal{N}_1}{q^{\ell_1+\cdots+\ell_t}} \left(\frac{\mathcal{N}_2}{q^{\ell_2+\cdots+\ell_t}} \right)^2 \cdots \left(\frac{\mathcal{N}_i}{q^{\ell_i+\cdots+\ell_t}} \right)^{2^{i-1}} > 1$$

for all $1 \leq i \leq t$. We now assume a best-case scenario, that all the splittings of all the $\mathcal{N}_{sol} \geq 1$ solutions are distinct (this is clearly unrealistic for large values of $\mathcal{N}_{sol}$, but it gives a rough idea of how much speedup one can ask with this approach). Then the success condition changes, for all $1 \leq i \leq t$, to

$$\mathcal{N}_{sol} \frac{\mathcal{N}_1}{q^{\ell_1+\cdots+\ell_t}} \left(\frac{\mathcal{N}_2}{q^{\ell_2+\cdots+\ell_t}} \right)^2 \cdots \left(\frac{\mathcal{N}_i}{q^{\ell_i+\cdots+\ell_t}} \right)^{2^{i-1}} > 1 \tag{4}$$

It follows that when $t = 3$ the best parameters $\ell_1, \ell_2, \ell_3, \alpha, \beta, \gamma$ are chosen by the following linear program:

$$\begin{aligned}
\text{minimize } & T = \max\left(\frac{\#(L^{(1)})^2}{q^{n-\ell_1-\ell_2-\ell_3}}, \frac{\#(L^{(2)})^2}{q^{\ell_1}}, \frac{\#(L^{(3)})^2}{q^{\ell_2}}, \sqrt{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}}\right) \\
\text{subject to } & \#L^{(1)} = \frac{\#\mathcal{X}_{1/4+\alpha,\alpha}}{q^{\ell_1+\ell_2+\ell_3}}, \\
& \#L^{(2)} = \frac{\#\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}}{q^{\ell_2+\ell_3}}, \\
& \#L^{(3)} = \frac{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}}{q^{\ell_3}}, \\
& \text{equation (4) holds for } 1 \leq i \leq 3 \\
& \ell_i \in \mathbb{N}, 1 \leq i \leq 3 \\
& \alpha, \beta, \gamma \in \mathbb{R}_{\geq 0}.
\end{aligned}$$

For the ISIS problem $\mathcal{B} = \{0,1\}^m$ given q and n, we can reduce m (i.e., guess some positions of the solution vector $\mathbf{x}$) to get lower density instances. Let the density be $2^{c_1 m}$ and the time complexity be $\tilde{O}(2^{c_2 m})$, the time complexity comes from choosing the optimal parameters α, β, γ and ℓ_1, ℓ_2, ℓ_3 for the given density. Figure 1 indicates how the density affects the asymptotic complexity for CPW, HGJ and BCJ. Thus we can see CPW is the best choice for high density instances, whereas HGJ/BCJ is suitable for low density instances. Further, it is reasonable to believe that a small speedup can be obtained with the HGJ and BCJ algorithms when running them on instances of density > 1. However, our analysis is based on some strong simplifying assumptions, and it should not be assumed that the HGJ and BCJ algorithms perform exactly this well when the density is moderate.

To invert the SWIFFT hash function the parameters are $\mathcal{B} = \{0,1\}^m$, $m = 1024$, $q = 257$, $n = 64$ and so the density is $2^{0.5m}$ which is a very high density instance. For this problem the CPW algorithm is the best choice.

4 Hermite normal form

We now give the main idea of the paper. For simplicity, assume that q is prime, $G = \mathbb{Z}_q^n$ and $n > 1$. We also assume that the matrix $\mathbf{A}$ has rank equal to n, which will be true with very high probability when $m \gg n$.

We exploit the Hermite normal form. Given an $n \times m$ matrix $\mathbf{A}$ over $\mathbb{Z}_q$ with rank $n < m$ then, by permuting columns as necessary, we may assume that $\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2]$ where $\mathbf{A}_1$ is an invertible $n \times n$ matrix and $\mathbf{A}_2$ is an $n \times (m-n)$ matrix. Then there exists a matrix $\mathbf{U} = \mathbf{A}_1^{-1}$ such that

$$\mathbf{UA} = [\mathbf{I}_n|\mathbf{A}']$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix and $\mathbf{A}'$ is the $n \times (m-n)$ matrix $\mathbf{UA}_2$. The matrix $[\mathbf{I}_n|\mathbf{A}']$ is called the Hermite normal form (HNF) of $\mathbf{A}$ and it can be computed (together with $\mathbf{U}$) by various methods. We assume q is prime and hence Gaussian elimination is sufficient to compute the HNF.

Writing $\mathbf{x} = \left(\frac{\mathbf{x}_0}{\mathbf{x}_1}\right)$ where $\mathbf{x}_0$ has length n and $\mathbf{x}_1$ has length $m-n$ we have that

$$\mathbf{s} \equiv \mathbf{Ax} \pmod{q} \quad \text{iff} \quad \mathbf{s}' = \mathbf{Us} \equiv \mathbf{A}'\mathbf{x}_1 + \mathbf{x}_0 \pmod{q}.$$

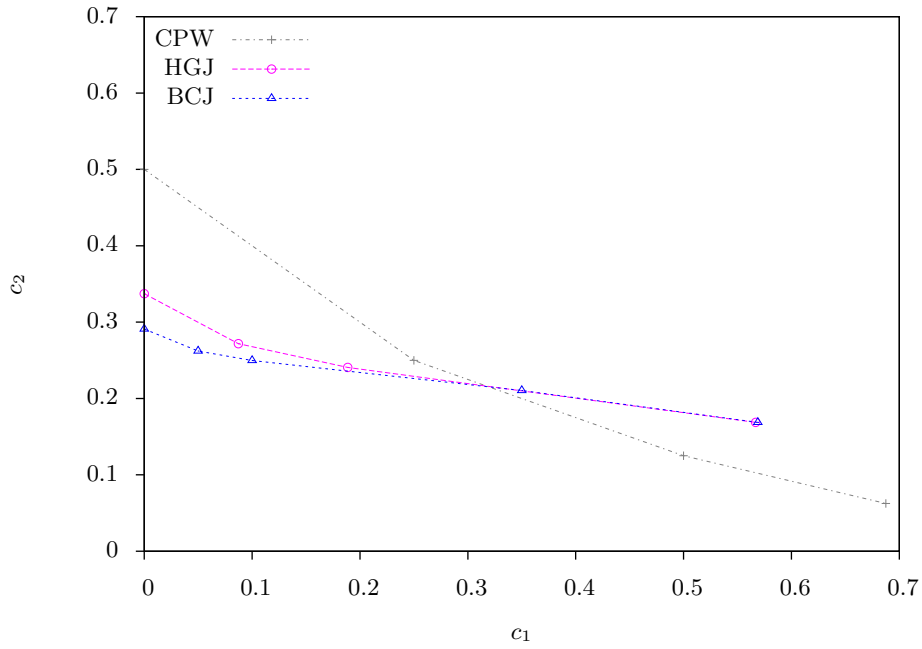

Fig. 1: Heuristic comparison of the performance of CPW, HGJ and BCJ algorithms on ISIS instances of density ≥ 1. The horizontal axis is the value c_1 such that the density (expected number of solutions) is $2^{c_1 m}$. The vertical axis is the constant c_2 such that the heuristic asymptotic complexity is $\tilde{O}(2^{c_2 m})$.

Hence, the Hermite normal form converts an ISIS instance to an instance of LWE (learning with errors) having bounded number of samples n.

It is not the goal of this paper to discuss the learning with errors problem in great detail. As with ISIS, it can be reduced to the closest vector problem in a lattice and hence solved using lattice basis reduction/enumeration techniques. There are two other notable algorithms for learning with errors, due to Arora-Ge and Blum-Kalai-Wasserman. However, since our variant of LWE has a fixed small number of samples they cannot be applied.

We will now apply the previous algorithms for the ISIS problem to this variant of the problem. This project was suggested in Section 6 of [11] to be an interesting problem (they called it the "approximate knapsack problem"). Our approach is to replace exact equality $\mathbf{y}_1 = \mathbf{y}_2$ of elements in quotient groups $G/H = \mathbb{Z}_q^\ell$, in certain parts of the algorithms, by an approximate equality $\mathbf{y}_1 \approx \mathbf{y}_2$. The definition of $\mathbf{y}_1 \approx \mathbf{y}_2$ will be that $\mathbf{y}_1 - \mathbf{y}_2 \in \mathcal{E}$, where $\mathcal{E}$ is some neighbourhood of 0. Different choices of $\mathcal{E}$ will lead to different relations, and the exact choice depends somewhat on the algorithm under consideration.

4.1 Approximate merge algorithm

Our main tool is to merge lists using an approximate algorithm. We write $\mathbf{Ax} \approx \mathbf{s}$ to mean $\mathbf{Ax} + \mathbf{e} = \mathbf{s}$ for some $\mathbf{e} \in \mathcal{E}$ in some set $\mathcal{E}$ of permitted errors. We warn the reader that this symbol $\approx$ is not necessarily an equivalence relation (e.g., it is not necessarily symmetric).

We use similar notation to Section 2.3: $\mathcal{X} \subseteq \mathcal{B}^m$ is a set of vectors, symbols H denote suitably chosen subgroups of G such that $\#(G/H_i) = q^{\ell_i}$. We split the set of vectors $\mathcal{X} \subseteq \mathcal{X}_1 + \mathcal{X}_2 = \{\mathbf{x}_1 + \mathbf{x}_2 : \mathbf{x}_1 \in \mathcal{X}_1, \mathbf{x}_2 \in \mathcal{X}_2\}$ in some way.

We also have a set of errors $\mathcal{E}$ and its splittings $\mathcal{E}_1, \mathcal{E}_2$. For example, we might take $\mathcal{E} = \mathcal{E}_1 = \mathcal{E}_2 = \{0,1\}^n$. Recall that we are trying to solve $\mathbf{s} \equiv \mathbf{A}\mathbf{x} + \mathbf{e}$ with $\mathbf{x} \in \mathcal{X}$ and $\mathbf{e} \in \mathcal{E}$. We also define the error sets $\mathcal{E}^{(i)}$ restricted to the quotient groups G/H_i, so that $\mathcal{E}^{(i)} = \{0,1\}^{\ell_i}$ or $\{-1,0,1\}^{\ell_i}$.

Let $H^\flat, H, H^\sharp$ be subgroups of G, the desired output of the merge algorithm is a set of solutions to the problem $\mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{s} \pmod{H \cap H^\flat}$ for $\mathbf{x} \in \mathcal{X}$ and $\mathbf{e} \in \mathcal{E}$, together with information about $\mathbf{A}\mathbf{x} \pmod{H^\sharp}$ to be used in future calculations. The input is a pair of lists L_1 and L_2 that are "partial solutions" modulo $H^\flat$. The details are given in Algorithm 2.

Algorithm 2 Approximate merge algorithm

INPUT: $L_1 = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod H) : \mathbf{A}\mathbf{x} + \mathbf{e} \equiv R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_1, \mathbf{e} \in \mathcal{E}_1\}$,
$L_2 = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod H) : \mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{s} - R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_2, \mathbf{e} \in \mathcal{E}_2\}$
OUTPUT: $L = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H^\sharp}) : \mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{s} \pmod{H \cap H^\flat}, \mathbf{x} \in \mathcal{X}, \mathbf{e} \in \mathcal{E}\}$
1: Initialise $L = \{\}$
2: Sort L_2 with respect to the second coordinate
3: **for** $(\mathbf{x}_1, \mathbf{u}) \in L_1$ **do**
4: Compute $\mathbf{v} = \mathbf{s} - \mathbf{u} \pmod H$
5: **for** $(\mathbf{x}_2, \mathbf{u}') \in L_2$ with $\mathbf{v} \approx \mathbf{u}'$ **do**
6: **if** $\mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{X}$ and $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \approx \mathbf{s} \pmod{H \cap H^\flat}$ **then**
7: Compute $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H^\sharp}$
8: Add $(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H^\sharp})$ to L

The detection of values $\mathbf{u}'$ in the sorted list such that $\mathbf{v} \approx \mathbf{u}'$ (meaning $\mathbf{v} + \mathbf{e} \approx \mathbf{u}'$ for some $\mathbf{e} \in \mathcal{E}^{(i)}$ can be done in several ways. One is to try all possible error vectors $\mathbf{e}$ and look up each candidate value $\mathbf{v} + \mathbf{e}$. Another is to "hash" using most significant bits. We give the details below. The running time of the algorithm depends on this choice.

Let $q^\ell = \#(G/H)$, for each pair $(\mathbf{x}_1, \mathbf{u}) \in L_1$ and each $\mathbf{e} \in \mathcal{E} \subseteq \mathbb{Z}_q^\ell$, the expected number of "matches" $(\mathbf{x}_2, \mathbf{u} + \mathbf{e})$ in L_2 is $\#L_2/q^\ell$. In the case where all values for $\mathbf{e} \in \mathcal{E}$ are chosen and then each candidate for $\mathbf{u}'$ is looked up in the table, then the running time is proportional to

$$\#L_1 \#\mathcal{E} \left\lceil \frac{\#L_2}{q^\ell} \right\rceil .$$

In the BCJ application in Section 4.3 we will take $\#\mathcal{E} = 3^\ell$ and we will always have $\#L_2 \geq q^\ell$. Hence the running time will be $\tilde{O}(\#L_1 \#L_2/(q/3)^\ell)$.

As previously, it is non-trivial to bound the size of the output list L. Instead, this can be bounded by $\#\mathcal{X}\#\mathcal{E}/\#(G/(H \cap H^\flat))$.

Note that different choices for $\mathcal{E}, \mathcal{E}_1, \mathcal{E}_2$ can lead to different organisation in the algorithm. For example, line 6 of Algorithm 2 involves a computation $\pmod{H \cap H^\flat}$, whereas this can be simplified to $\pmod H$ if the errors are chosen appropriately – this is what we will do when adapting the CPW algorithm to this case.

4.2 The CPW algorithm

Recall that our problem is (taking the HNF and then renaming $(\mathbf{A}', \mathbf{s}')$ as $(\mathbf{A}, \mathbf{s})$): Given $\mathbf{A}, q, \mathbf{s}$ to solve $\mathbf{A}\mathbf{x}_1 + \mathbf{x}_0 = \mathbf{s}$ in $G = \mathbb{Z}_q^n$, where $\mathbf{x}_1$ has length $m - n$ and $\mathbf{x}_0$ has length n. We assume the problem has high density, in the sense that there are many pairs $(\mathbf{x}_0, \mathbf{x}_1) \in \mathcal{B}^m$ that solve the system.

As we have seen, the CPW algorithm is most suitable for problems with very high density, since higher density means more lists can be used and so the running time is lower. Hence, it may seem that reducing m to $m-n$ will be unhelpful for the CPW algorithm. However, we actually get a very nice tradeoff. In some sense, the high density is preserved by our transform while the actual computations are reduced due to the reduction in the size of m.

As noted, we define a (not-necessarily symmetric or transitive) relation $\approx$ on vectors in $G = \mathbb{Z}_q^n$ as $\mathbf{v} \approx \mathbf{w}$ if and only if $\mathbf{v} - \mathbf{w} \in \mathcal{B}^n$. One can then organise a CPW-style algorithm: compute the lists $L_j^{(i)}$ as usual, but merge them using $\approx$. However, we need to be a bit careful. Consider the case of 4 lists. Lists $L_j^{(0)}$ contain pairs $(\mathbf{x}, \mathbf{Ax})$ (in the case of $L_4^{(0)}$ it is $(\mathbf{x}, \mathbf{Ax}-\mathbf{s})$). Merging $L_1^{(0)}$ and $L_2^{(0)}$ gives a list $L_1^{(1)}$ of pairs $(\mathbf{x}_1, \mathbf{x}_2)$ such that $\mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) \approx 0 \pmod{H_1}$, which means $\mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) = \mathbf{e}$ for some $\mathbf{e} \in \mathcal{B}^{n/3}$. Similarly, $L_2^{(1)}$ is a list of pairs $(\mathbf{x}_3, \mathbf{x}_4)$ such that $\mathbf{A}(\mathbf{x}_3+\mathbf{x}_4) = \mathbf{e}'$ for some $\mathbf{e}' \in \mathcal{B}^{n/3}$. The problem is that $\mathbf{e}+\mathbf{e}'$ does not necessarily lie in $\mathcal{B}^{n/3}$ and so the merge at the final stage will not necessarily lead to a solution to the problem.

There are several ways to avoid this issue. One would be to "weed out" these failed matches at the later stages. However, our approach is to constrain the relation $\approx$ further during the merge operations. Specifically (using the notation of the previous paragraph) we require the possible non-zero positions in $\mathbf{e}$ and $\mathbf{e}'$ to be disjoint.

The details To be precise, let $k = 2^t$ be the number of lists. We define $u = (m-n)/k$ to be the number of entries in each $\mathbf{x}_j$ and let $\mathcal{X}_j = \{(0, \ldots, 0, x_{(j-1)u+1}, \ldots, x_{ju}, 0, \ldots, 0) \in \mathcal{B}^m\}$ for $1 \le j \le k$. In turns out to be better to not have all merges using quotient groups of the same size, so we choose integers $\ell_i > 0$ such that $\ell_1 + \ell_2 + \cdots + \ell_t = n$. We will choose the subgroups H_i so that $G/H_i \cong \mathbb{Z}_q^{\ell_i}$ for $1 \le i \le t$. So $H_1 = \{(0, 0, \cdots, 0, g_{\ell_1+1}, \cdots, g_n) \in \mathbb{Z}_q^n\}$, $H_2 = \{(g_1, \ldots, g_{\ell_1}, 0, \ldots, 0, g_{\ell_1+\ell_2+1}, \ldots, g_n)\}$ and so on.

For parameters $k' \in \mathbb{N}, \gamma$ we define error sets in $\mathcal{B}^{k'\gamma}$ for $1 \le j \le k'$ as

$$\mathcal{E}_{\gamma,j} = \{(0, \ldots, 0, e_{(j-1)\gamma+1}, \ldots, e_{j\gamma}, 0, 0, \ldots, 0) \in \mathcal{B}^{k'\gamma}\}.$$

Note that $\#\mathcal{E}_{\gamma,j} = (\#\mathcal{B})^\gamma$.

Level 0: Compute lists $L_j^{(0)} = \{(\mathbf{x}, \mathbf{Ax} \pmod{H_1}) \in \mathcal{X}_j \times \mathbb{Z}_q^{\ell_1}\}$ for $1 \le j \le k-1$ and $L_k^{(0)} = \{(\mathbf{x}, \mathbf{Ax}-\mathbf{s} \pmod{H_1}) \in \mathcal{X}_k \times \mathbb{Z}_q^{\ell_1}\}$. Note that $\#L_j^{(0)} = \#\mathcal{B}^u = \#\mathcal{B}^{(m-n)/k}$. The cost to compute the initial k lists is approximately $k \cdot \#L_j^{(0)} \cdot C$, where C is the number of bit operations to compute a sum of at most u vectors in $\mathbb{Z}_q^{\ell_1}$ i.e. $C = (m-n)\log_2(q^{\ell_1})/k$.

Level 1: We now merge the $k = 2^t$ lists in pairs to get $k/2 = 2^{t-1}$ lists. Let $\gamma_1 = \ell_1/(k/2)$. The sets $\mathcal{E}_{\gamma_1,j}$ specify the positions that are allowed to contain errors. In other words, for $j = 1, 2, \cdots, k/2$ we construct the new lists

$$L_j^{(1)} = \{(\mathbf{x}_1, \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) \pmod{H_2}) : \mathbf{x}_1 \in L_{2j-1}^{(0)}, \mathbf{x}_2 \in L_{2j}^{(0)}, \mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) \pmod{H_1} \in \mathcal{E}_{\gamma_1,j}\}.$$

The probability that two random vectors in $\mathbb{Z}_q^{\ell_1}$ have sum in $\mathcal{E}_{\gamma_1,j}$ is $\#\mathcal{E}_{\gamma_1,j}/q^{\ell_1} = \#\mathcal{B}^{\gamma_1}/q^{\ell_1}$, and so the expected size of the lists $L_j^{(1)}$ is $\#L_{2j-1}^{(0)}\#L_{2j}^{(0)}\#\mathcal{B}^{\gamma_1}/q^{\ell_1} \approx \#\mathcal{B}^{2(m-n)/k+\gamma_1}/q^{\ell_1}$, which we will want to be roughly $\#\mathcal{B}^{(m-n)/k}$ again.

Level $i \ge 2$: The procedure continues in the same way. We are now merging $k/2^{i-1}$ lists to get $k/2^i$ lists. We do this by checking ℓ_i coordinates and so will allow errors for each merge in only

$\gamma_i = \ell_i/(k/2^i)$ positions. Hence, for $j = 1, 2, \cdots, k/2^i$ we construct the new lists

$$L_j^{(i)} = \{(\mathbf{x}_1, \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) \pmod{H_{i+1}}) : \mathbf{x}_1 \in L_{2j-1}^{(i-1)}, \mathbf{x}_2 \in L_{2j}^{(i-1)}, \mathbf{A}(\mathbf{x}_1+\mathbf{x}_2) \pmod{H_i} \in \mathcal{E}_{\gamma_i,j}\}.$$

As before, the expected size of $L_j^{(i)}$ is $\#L_{2j-1}^{(i-1)} \# L_{2j}^{(i-1)} \# \mathcal{B}^{\gamma_i}/q^{\ell_i}$.

It remains to explain how to perform the merging of the lists. The problem is that we cannot simply do the usual "sort and match" process as this will only find exact matches rather than matches up to an error. We are seeking a match on vectors in $\mathbb{Z}_q^{\ell_i}$ that are equal on all but γ_i coordinates, and that are "close" on those γ_i coordinates. The natural solution is to detect matches using the most significant bits of the coordinates (this approach was used in a similar situation by Howgrave-Graham, Silverman and Whyte [10]). Precisely, let v be a parameter (indicating the number of most significant bits being used). Represent $\mathbb{Z}_q$ as $\{0, 1, \ldots, q-1\}$ and define a hash function $F : \mathbb{Z}_q \to \mathbb{Z}_{2^v}$ by $F(x) = \lfloor \frac{x}{q/2^v} \rfloor$. We can then extend F to $\mathbb{Z}_q^{\gamma_i}$ (and to the whole of $\mathbb{Z}_q^{\ell_i}$ by taking the identity map on the other coordinates). We want to detect a match of the form $\mathbf{A}\mathbf{x}_1 + \mathbf{A}\mathbf{x}_2 + \mathbf{e} = 0$, which we will express as $\mathbf{A}\mathbf{x}_1 = -\mathbf{A}\mathbf{x}_2 - \mathbf{e}$. The idea is to compute $F(\mathbf{A}\mathbf{x}_1)$ for all $\mathbf{x}_1$ in the first list and store these in a sorted list. For each value of $\mathbf{x}_2$ in the second list one computes all possible values for $F(-\mathbf{A}\mathbf{x}_2 - \mathbf{e})$ and checks which of them are in the sorted list.

For example, consider $q = 2^3 = 8$ and suppose we use a single most significant bit (so $F : \mathbb{Z}_q \to \{0, 1\}$). Suppose $\mathbf{A}\mathbf{x}_1 = (7, 2, 3, 5, 6, 4, 0, 7)^T$ and that we are only considering errors on the first $\gamma = 4$ coordinates. Then we have $F(\mathbf{A}\mathbf{x}_1) = (1, 0, 0, 1, 6, 4, 0, 7)$. Suppose now $-\mathbf{A}\mathbf{x}_2 = (7, 3, 4, 5, 6, 4, 0, 7)$. Then $F(-\mathbf{A}\mathbf{x}_2) = (1, 0, 1, 1, 6, 4, 0, 7)$. By looking at the "borderline" entries of $-\mathbf{A}\mathbf{x}_2$ we know that we should also check $(1, 0, 0, 1, 6, 4, 0, 7)$. There is no other value to check, since $F((7, 3, 4, 5) - (1, 1, 1, 1)) = F(6, 2, 3, 4) = (1, 0, 0, 1)$ and so the only possible values for the first 4 entries of $F(-\mathbf{A}\mathbf{x}_2 - \mathbf{e})$ are $\{(1, 0, 0, 1), (1, 0, 1, 1)\}$.

To be precise we define $\text{Flips}(\mathbf{v}) = \{F(\mathbf{v} - \mathbf{e}) : \mathbf{e} \in \mathcal{E}_{\gamma,j}\}$, where j and γ_i are clear in any given iteration of the algorithm. In other words, it is the set of all patterns of most significant bits that would arise by adding valid errors to the corresponding coordinates of $\mathbf{v}$. The set $\text{Flips}(\mathbf{v})$ is not likely to be large, since it only arises when the vector has some coordinates that are exactly on the borderline. Let p_{flip} be the probability that a randomly chosen element of $\mathbb{Z}_q$ has hash value that flips when subtracting an error.

1. If $\mathcal{B} = \{0, 1\}$ then $p_{\text{flip}} = 2^v/q$. Thus, on average, $\#\text{Flips}(\mathbf{v}) = 2^{\gamma_i 2^v/q}$.
2. If $\mathcal{B} = \{-1, 0, 1\}$ then $p_{\text{flip}} = 2^{v+1}/q$. Thus, on average, $\#\text{Flips}(\mathbf{v}) = 2^{\gamma_i 2^{v+1}/q}$.

To summarise the "approximate 2-merge" algorithm: First compute $F(\mathbf{v})$ for every $\mathbf{v} = \mathbf{A}\mathbf{x}_1$ in the list $L_{2j-1}^{(i-1)}$, and sort these values. Note that there may be multiple different values $\mathbf{x}_1$ in the list with the same hash value $F(\mathbf{A}\mathbf{x}_1)$. Then, for every $\mathbf{v} = -\mathbf{A}\mathbf{x}_2$ for $\mathbf{x}_2$ in the list $L_{2j}^{(i-1)}$ we compute $\text{Flips}(\mathbf{v})$ and search for a match in the sorted list. Finally, for each match, we go through all values $\mathbf{x}_1$ in the first list with the given hash value $F(\mathbf{A}\mathbf{x}_1)$ and, for each of them, check if it really is true that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2)$ is in the correct error set (since a match of the hash values does not imply correctness). The number of possible hash values on vectors in $\mathbb{Z}_q^{\ell_i}$, with γ_i positions reduced to the v most significant bits, is $2^{v\gamma_i} q^{\ell_i - \gamma_i}$. Hence, the average number of values in the list $L_{2j-1}^{(i-1)}$ that take a given hash value is $\#L_{2j-1}^{(i-1)}/(2^{v\gamma_i} q^{\ell_i - \gamma_i})$. Finally, for all good matches we need to compute $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_i}$.

The average total time for the approximate merge algorithm is therefore (cost of sorting plus cost of searching plus cost of consistency check plus cost of recomputing)

$$\#L_{2j-1}^{(i-1)}\left(C_1+\log_2(\#L_{2j-1}^{(i-1)})\log_2(q^{\ell_i})\right) \;+\; \#L_{2j}^{(i-1)}\left(C_2+2^{\gamma_i p_{\text{flip}}}\left[\log_2(\#L_{2j-1}^{(i-1)})\log_2(q^{\ell_i})+ (\#L_{2j-1}^{(i-1)}/(2^{v\gamma_i}q^{\ell_i-\gamma_i}))C_3\right]\right)+\; \#L_j^{(i)}(2\log_2(q^{\ell_{i+1}})) \tag{5}$$

where C_1 and C_2 are the cost of computing F for the γ_i positions which allow errors, and C_3 is the cost of checking that $\mathbf{x}_1+\mathbf{x}_2\in\mathcal{B}^m$ and that $\mathbf{A}(\mathbf{x}_1+\mathbf{x}_2)\in\mathcal{E}$ (we only need to check γ_i positions of the error). Larger values for v increase p_{flip} but reduce $\#L_{2j-1}^{(i-1)}/(2^{v\gamma_i}q^{\ell_i-\gamma_i})$. So we choose v to balance the costs $2^{\gamma_i p_{\text{flip}}}$ and $\#L_{2j-1}^{(i-1)}/(2^{v\gamma_i}q^{\ell_i-\gamma_i})$. Hence, the time to do "approximate 2-merge" can be proportional to $\#L_{2j-1}^{(i-1)}\log_2(\#L_{2j-1}^{(i-1)})$. When $\gamma_i=0$, actually it's the basic merge algorithm and so $C_1=0, C_2=0, C_3=u$, $v=\log_2(q), 2^{\gamma_i p_{\text{flip}}}=1, \#L_{2j-1}^{(i-1)}/(2^{v\gamma_i}q^{\ell_i-\gamma_i})=\#L_{2j-1}^{(i-1)}/q^{\ell_i}$.

To make sure the algorithm can find one solution at the bottom level, we require $\#L_1^{(t)}$ to have expected size at least one. Denote by $L^{(i)}$ any of the lists at the i-th stage of the algorithm. Recall that we want to minimise $\max_{0\le i\le t}(\#L^{(i)})$ and that we have

$$\#L^{(i)}\le \#L^{(i-1)}\#L^{(i-1)}\#\mathcal{B}^{\gamma_i}/q^{\ell_i}. \tag{6}$$

Since $\#L^{(0)}=(\#\mathcal{B})^{(m-n)/k}$ we can write $\#L^{(i)}=2^{b_i}$ where $b_0=\frac{m-n}{k}\log_2(\#\mathcal{B})$. It is usually the case that equation (6) is an equality, and hence get $b_i=2b_{i-1}-\log_2(q)\ell_i+\gamma_i\log_2(\#\mathcal{B})$. So to minimize the time complexity the ℓ_i should be a solution of the following integer program:

$$\begin{aligned}
\text{minimize}\quad & b_{max}=\max_{0\le i\le t} b_i\\
\text{subject to}\quad & 0\le b_i\le b_{max}, \qquad 0\le i\le t\\
& b_0=(m-n)\log_2(\#\mathcal{B})/k,\\
& b_i=2b_{i-1}-\log_2(q)\ell_i+\gamma_i\log_2(\#\mathcal{B}),\\
& \ell_i\ge 0, \qquad 0\le i\le t\\
& \sum_{i=1}^{t}\ell_i=n.
\end{aligned}$$

Given m,n,q, to reduce the time and space complexity we want k to be as large as possible. However, as with the original Wagner algorithm, to make sure the expected number of solutions found is non-zero, k can't be too large. In practice, the algorithm chooses the largest $k=2^t$ that guarantees $\#L_1^{(t)}\approx 1$. Note that the higher the density of the (I)SIS problem, the larger k can be chosen.

4.3 HGJ and BCJ for approximate-ISIS problem

The HGJ and BCJ algorithms can be implemented to solve the approximate-ISIS problem $\mathbf{Ax}+\mathbf{e}\equiv\mathbf{s}\pmod q$, (where $\mathbf{A}$ is a $n\times m'$ matrix, $\mathbf{x}\in\{0,1\}^{m'}$, $\mathbf{e}\in\{0,1\}^n$) by using Algorithm 2. Assume for simplicity that the density is approximately 1 i.e. $2^{m'+n}/q^n\approx 1$. Since HGJ is a special case of BCJ, we discuss the general case.

In the context of the Hermite normal form of an (m, n)-ISIS instance, the best outcome we could expect would be an algorithm with heuristic complexity $\tilde{O}(2^{0.291m'})$ where $m' = m - n$. We believe one can get close to this by choosing suitable parameters, but there are a number of complications in the analysis. We give a possible choice of the parameters to realize this time complexity, however the success probability is hard to analyse.

We use the BCJ algorithm with 3 levels of recursion. Recall that $\mathcal{X}_{a,b}$ denotes vectors in $\{-1, 0, 1\}^{m'}$ with am' entries equal to 1 and bm' entries equal to -1. Recall that we are trying to solve $\mathbf{s} \equiv \mathbf{A}\mathbf{x} + \mathbf{e}$ with $\mathbf{x} \in \mathcal{X}$ and $\mathbf{e} \in \{0, 1\}^n$. Write coordinates of $\mathbf{e}$ as e_i, using BCJ's idea to split the error, if coordinate $e_i = 0$ then it can be split in three ways as $-1 + 1$, $0 + 0$ or $1 + -1$, while if $e_i = 1$ then it can only be split in two ways $1 + 0$ and $0 + 1$. For example, the first level of recursion splits $\mathcal{X} = \mathcal{X}_{1/2,1/2} \subset \mathcal{B}^{m'}$ into $\mathcal{X}_1 + \mathcal{X}_2$ where $\mathcal{X}_1 = \mathcal{X}_2 = \mathcal{X}_{1/4+\alpha,\alpha}$, and also split the error sets $\{0, 1\}^n$ into two sets $\mathcal{E}_1 = \mathcal{E}_2 = \{-1, 0, 1\}^n$. In other words, we consider splitting $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ and $\mathbf{e} = \mathbf{e}_1 + \mathbf{e}_2$ with $\mathbf{x}_i \in \mathcal{X}_i$ and $\mathbf{e}_i \in \mathcal{E}_i = \{-1, 0, 1\}^n$. Note that not every value in $\mathcal{E}_i$ arises as a possible splitting of the target solution $(\mathbf{x}, \mathbf{e})$. So we define the error sets $\mathcal{E}^{(i)}$ restricted to the quotient groups G/H_i, so that $\mathcal{E}^{(i)} = \{-1, 0, 1\}^{\ell_i}$ using the notation from Section 4.1. Define the error set $\mathcal{E}^{(0)} = \{0, 1\}^n$ restricted to the group G. Further, we define $\mathcal{E}^{(i\to t)} = \{-1, 0, 1\}^{\ell_i+\cdots+\ell_t}$ restricted to the quotient groups $G/(H_i \cap \cdots H_t)$. To use Algorithm 2, the simplest choice here is to take $\mathcal{E}_1 = \mathcal{E}_2 = \{-1, 0, 1\}^{\ell_i}$ when solving equations modulo H_i then"weed out" the inconsistent errors at later stages.

In the first level of recursion, we compute two lists $L_1^{(1)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_1, \mathbf{A}\mathbf{x} + \mathbf{e} \equiv R_1 \pmod{H_1 \cap H_2 \cap H_3}$ for some $\mathbf{e} \in \{-1, 0, 1\}^{\ell_1+\ell_2+\ell_3}\}$ and $L_2^{(1)}$ which is that same except $\mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{s} - R_1 \pmod{H_1 \cap H_2 \cap H_3}$. The expected size of the lists is $\#L^{(1)} = \#\mathcal{X}_1 \#\mathcal{E}^{(1\to3)}/(q)^{\ell_1+\ell_2+\ell_3} \approx 2^{H(1/4+\alpha,\alpha)m'}/(q/3)^{\ell_1+\ell_2+\ell_3}$ and merging requires $\tilde{O}(\#L^{(1)}\#L^{(1)}(2/q)^n/(3/q)^{\ell_1+\ell_2+\ell_3})$ time.

The second level of recursion computes each of $L_1^{(1)}$ and $L_2^{(1)}$, by splitting into further lists. We split $\mathcal{X}$ to $\mathcal{X}_1 = \mathcal{X}_2 = \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$ and use error vectors $\mathbf{e} \in \mathcal{E}^{(2\to3)} = \{-1, 0, 1\}^{\ell_2+\ell_3}$.

For example, $L_1^{(1)}$ is split into $L_1^{(2)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_1, \mathbf{A}\mathbf{x}+\mathbf{e} \equiv R_2 \pmod{H_2 \cap H_3}\}$ and $L_2^{(2)}$ is similar except the congruence is $\mathbf{A}\mathbf{x}+\mathbf{e} \equiv R_1 - R_2 \pmod{H_2 \cap H_3}$. Again, the size of lists is approximately $\#L^{(2)} = 2^{H(1/8+\alpha/2+\beta,\alpha/2+\beta)m'}/(q/3)^{\ell_2+\ell_3}$ and the cost to merge is $\tilde{O}(\#L^{(2)}\#L^{(2)}/(q/3)^{\ell_1})$.

The final level of recursion computes each $L_j^{(2)}$ by splitting into two lists corresponding to coefficient sets $\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}$, and again error vectors $\mathbf{e} \in \{-1, 0, 1\}^{\ell_3}$. The expected size of the lists is $\#L^{(3)} = 2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m'}/(q/3)^{\ell_3}$ and they can be computed using the Shroeppel-Shamir algorithm in time $\tilde{O}(\sqrt{2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m'}})$. Merging the lists takes $\tilde{O}(\#L^{(3)}\#L^{(3)}/(q/3)^{\ell_2})$ time.

Let, as before, $\alpha = 0.0267, \beta = 0.0168$ and $\gamma = 0.0029$, and take integers ℓ_1, ℓ_2, ℓ_3 such that $(q/3)^{\ell_1} \approx 2^{0.267m'}, (q/3)^{\ell_2} \approx 2^{0.291m'}$ and $(q/3)^{\ell_3} \approx 2^{0.241m'}$. Choose subgroups H_1, H_2, H_3 such that $\#(G/H_i) = q^{\ell_i}$. Then the time complexity is $\tilde{O}(2^{0.291m'})$.

The algorithm succeeds if there is a splitting of the desired solution $(\mathbf{x}, \mathbf{e})$ to $\mathbf{A}\mathbf{x} + \mathbf{e} = \mathbf{s}$ in $\mathbb{Z}_q^n$ (assumed now to be unique) that satisfies all the "approximate" congruences in the recursion. In this case, we need to work out how many ways there are to split $(\mathbf{x}, \mathbf{e})$ in each level of recursion. Counting the ways to split $\mathbf{x}$ is similar to BCJ's analysis in Section 2.8, while the counting the ways to split $\mathbf{e}$ is harder to work out exactly. For example, to analyse the ways to split in the first level of the recursion $\mathbf{e} \equiv \mathbf{e}_1 + \mathbf{e}_2 \pmod{H_1 \cap H_2 \cap H_3}$, we need to know the numbers of 0 and 1 in the $\ell_1 + \ell_2 + \ell_3$ coordinates of $\mathbf{e} \in \{0, 1\}^n$. Since 0 can be split in three ways as $-1 + 1$, $0 + 0$, or $1 + -1$, while 1 only can be split in two ways as $1 + 0$ and $0 + 1$. Now consider the second level

of the recursion $\mathbf{e}_1 \equiv \mathbf{e}_{11} + \mathbf{e}_{12} \pmod{H_2 \cap H_3}$, we need to know the numbers of -1, 0 and 1 in the $\ell_2 + \ell_3$ coordinates of $\mathbf{e}_1 \in \{-1, 0, 1\}^{\ell_1+\ell_2+\ell_3}$. Since 0 has three ways to split, while -1 and 1 only have two ways to split. In other words, when considering the later level of recursion, we should know the probability distribution from the earlier level of recursion. So, it is hard to give a general analysis.

5 Improved attacks on SWIFFT

We propose improved attacks for both inversion and collision of SWIFFT. The standard SWIFFT problem uses parameters $m = 1024, n = 64, q = 257$ and binary secrets. The best previously known attacks for SWIFFT inversion and collision problem have asymptotic time 2^{148} bit operations and 2^{113} bits operations. In summary, our improved attacks solve the SWIFFT inversion and collision problems in asymptotic running time 2^{138} bit operations and 2^{104} bit operations.

5.1 Inverting SWIFFT.

Lyubashevsky, Micciancio, Peikert and Rosen [16] discussed solving the $(1024, 64, 257, \{0, 1\})$-SIS problem using the original CPW algorithm: "it is also possible to mount an inversion attack using time and space approximately 2^{128}". They choose $k = 8$ to break up the 1024 column vectors of matrix $\mathbf{A}$ into 8 groups of 128 column vectors each. For each group create a list of size 2^{128}, then choose $\ell_1 = 16, \ell_2 = 16, \ell_3 = 32$, at each level the size of the lists is around 2^{128}, so the required storage is $8 \cdot 2^{128} \log_2(q^{16})$ bits. Using the formula (5), we predict the total running time to be approximately 2^{148} bit operations.

We now show that using the HNF trick and our approximate CPW algorithm from section 4.2 gives a 2^{10} speed-up. First, we reduce the dimension from 1024 to 1000 by setting 24 entries of $\mathbf{x}$ to be zero and deleting the corresponding columns from $\mathbf{A}$. Then compute the Hermite normal form, to reduce $\mathbf{A}$ to a 64×936 matrix. We then use $k = 8$ to break split $\{0, 1\}^{936}$ into 8 groups of length 117. Let $\ell_1 = 15, \ell_2 = 16, \ell_3 = 33$. Construct 8 initial lists of size 2^{117}. At each step, we merge two lists in a similar way to the original CPW algorithm. However, to find approximate collisions we use the "approximate merge" algorithm described in Algorithm 2.

Level 1, we merge the initial 8 lists of size 2^{117} by checking the first $\ell_1 = 15$ coordinates of the vectors. We allow errors in $\gamma_1 = 4, 4, 3, 4$ positions for each merge. The expected size of the three new lists corresponding to $\gamma_1 = 4$ is $\frac{2^{2 \cdot 117} \cdot 2^4}{257^{15}} \approx 2^{117.92}$, and the expected size of the other list is $\frac{2^{2 \cdot 117} \cdot 2^3}{257^{15}} \approx 2^{116.92}$.

For the hashing, we take $v = 7$ most significant bits of each value in $\mathbb{Z}_{257}$. The probability $p_{\text{flip}} \approx 0.5$, $2^{\gamma_1 p_{\text{flip}}} \leq 4$ and $\frac{2^{117}}{2^{v\gamma_1} q^{\ell_1 - \gamma_1}} \leq 2$.

Level 2, we merge the 4 lists on level 1 of sizes $2^{117.92}$, $2^{117.92}$, $2^{116.92}$, $2^{117.92}$ by checking the next $\ell_2 = 16$ coordinates of the vectors. We allow errors in $\gamma_2 = 8$ positions for each merge. The expected sizes of the 2 new lists is $\frac{2^{117.92+117.92} \cdot 2^8}{257^{16}} \approx 2^{115.75}$ and $\frac{2^{116.92+117.92} \cdot 2^8}{257^{16}} \approx 2^{114.75}$. For the hashing of each merge, we use $v = 7$.

Level 3, we now merge the 2 lists on level 2 of size $2^{114.75}$ and $2^{115.75}$ by checking the remaining $\ell_3 = 33$ coordinates of the vectors, allowing $\gamma_3 = 33$ positions to have errors. The expected size of the solution set[6] is $\frac{2^{114.75+115.75} 2^{33}}{257^{33}} \approx 2^{-0.7}$, we use $v = 4$ for the hashing.

[6] It's possible to tune certain constraints of the integer program in section 4.2 to get a better attack. Here we tune the constraint $b_t \geq 0$ to be $b_t \geq -1$, which means we expect to run the whole algorithm two times.

As a conclusion, the maximum size of the lists on each level is $2^{117.92}$, and using formula (5) we estimate the total time to be around 2^{138} bit operations.

5.2 Finding collisions for SWIFFT.

Finding collisions for SWIFFT is equivalent to solving the $(1024, 64, 257, \{-1, 0, 1\})$-SIS problem. Lyubashevsky, Micciancio, Peikert and Rosen [16] give an analysis using the CPW algorithm and choosing $k = 16$. They break up the 1024 column vectors of $\mathbf{A}$ into 16 groups of 64 vectors each, for each group create an initial list of $3^{64} \approx 2^{102}$ vectors. They choose $\ell_1 = \ell_2 = \ell_3 = \ell_4 = 13$ to perform the merges. They very optimistically assume that, at each level, the lists have 2^{102} vectors, and at the final level they end up with a list of $\approx 2^{102}$ elements whose first 52 coordinates are all zero. Since $2^{102} > 257^{12} \approx 2^{96}$, it is expected that there exists an element whose last 12 coordinates are also zero, they say "the space is at least 2^{102}, the running time is at least 2^{106}".

However, the assumption in [16] that the lists have 2^{102} elements at each level is implausible (but this is permitted in their context, since their goal is a lower bound on the running time). In fact the lists get smaller and smaller (sizes $2^{102} \to 2^{100} \to 2^{96} \to 2^{88} \to 2^{72}$) and so one does not have a list of 2^{102} vectors at the final level. Indeed, the success probability of their algorithm is only around 2^{-24}, and so running the algorithm 2^{23} times brings the running time to be about 2^{144} bit operations.

One can resolve the success probability issue by using Minder and Sinclair's refinement of CPW [19]. For $k = 16$ lists one can take $\ell_1 = 12, \ell_2 = 14, \ell_3 = 12, \ell_4 = 26$ and the maximum size of the lists at all the levels is around 2^{107}. Using formula (5) we estimate the total time to be about 2^{126} bit operations.

Howgrave-Graham and Joux described an improved collision attack in Appendix B of an early version of [12]. The idea is to attack the original $\{0, 1\}$-SIS problem directly: first using the original CPW algorithm to get a list of elements with a certain subset of their coordinates equal to 0, then exploit the birthday paradox using the elements in this list to find a collision between the remaining coordinates. They choose $k = 16$ and create 16 initial lists of size 2^{64}, choosing $\ell_1 = 4, \ell_2 = 12, \ell_3 = 12, \ell_4 = 12$, then the size of the lists on each level is 2^{96}. At the final step they obtain a list of 2^{96} elements with the first 40 coordinates equal to zero. Since $(2^{96})^2 \approx 257^{24}$, the birthday paradox shows one can find a collision between the remaining $24 = n - (\ell_1 + \ell_2 + \ell_3 + \ell_4)$ coordinates in this list. In other words, we have $\mathbf{A}\mathbf{x}_1 \equiv \mathbf{A}\mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \{0, 1\}^m$ and so we have found a collision for SWIFFT. The space requirement is about 2^{96} and the time is predicted in [12] to be proportional to 2^{100}. Formula (5) suggests the total time is about 2^{113} bit operations; a speedup by 2^{13} from the Minder-Sinclair method.

We now describe a better collision attack, by using our HNF trick and our approximate-CPW algorithm from Section 4.2. We apply the Hermite normal form to have an $n \times m'$ instance, where $m' = m - n = 960$. We then apply the CPW algorithm to construct a list of $\mathbf{x} \in \{0, 1\}^{960}$ such that $\mathbf{A}\mathbf{x}$ has a fraction of coordinates lying in $\{-1, 0\}$. Finally we exploit the birthday paradox to find a near collision between the remaining coordinates (here "near collision" means that the difference of the coordinates lies in $\{-1, 0, 1\}$).

Let $k = 16$ and break up the matrix into 16 groups of 60 vectors each. For each group create an initial list. We can control the size of the initial lists, as long as they are smaller than 2^{60}. The initial lists don't need to have the same size. We choose $\ell_1 = 5, \ell_2 = 10, \ell_3 = 11, \ell_4 = 12$ to perform our approximate merge. These values can be obtained by solving the integer program described in Section 4.2, we only need to change the constraint $b_t >= 0$ (one solution survives at the bottom

level) of the integer program in Section 4.2 to be $2b_t + \log_2(3) \cdot (n - \sum_{i=1}^{t} \ell_i) \geq \log_2(q) \cdot (n - \sum_{i=1}^{t} \ell_i)$, i.e. on the last level we want the size of the list is large enough to exploit birthday paradox. As long as this size is sufficiently large, there exist two elements (a near collision) $\mathbf{x}_1, \mathbf{x}_2$ such that $\mathbf{A}(\mathbf{x}_1 - \mathbf{x}_2)$ has its remaining coordinates all coming from $\{-1, 0, 1\}$. Figure 2 shows the size of the lists in each level and other parameters. We eventually obtain a list of $2^{83.45}$ elements with 38 coordinates equal to $\{-1, 0\}$. Since $2^{83.45+83.45} 3^{26} \approx 257^{26}$, obtaining a list of size $2^{83.45}$ in the final step of CPW is large enough to exploit the birthday paradox.

In summary, the maximum size of the lists is $2^{83.88}$, then the space is proportional to 2^{84}. By formula (5) the total running time is estimated to be 2^{104} bit operations; a 2^9 speed-up over the previous best method.

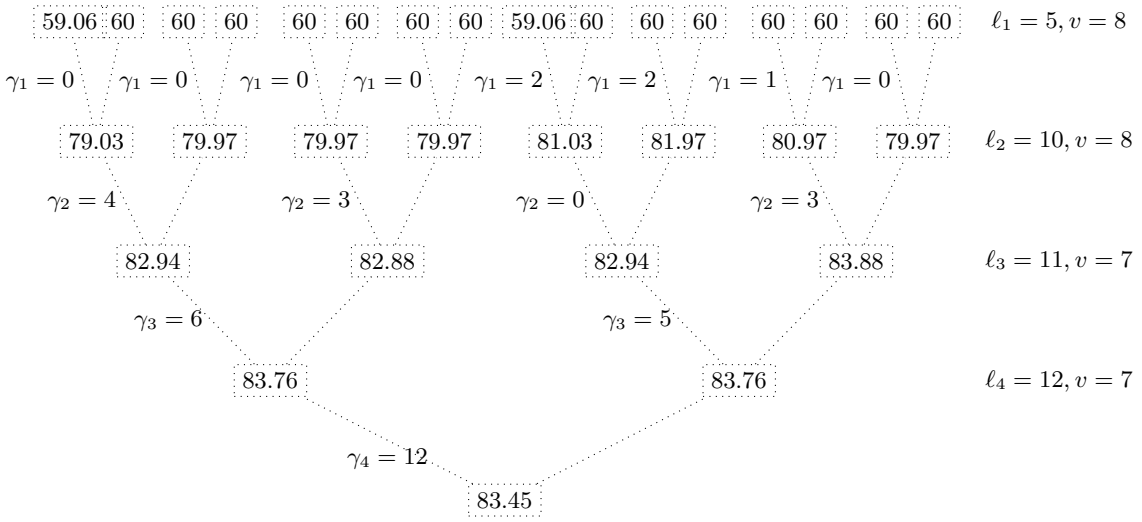

Fig. 2: Parameter choices and list sizes for the approximate-CPW algorithm for finding collisions in SWIFFT. The numbers in the dotted box denote the $\log_2$ size of the list; the γ_i is used in the approximate-merge algorithm; v is the number of most significant bits we use for the hash in the cases when $\gamma_i \neq 0$.

6 Experimental results

The purpose of this section is two-fold: (1) to show our size and complexity estimate is robust: the actual running-time of the algorithm follows from the bit complexity (and hence size) estimate; (2) to show that our improved algorithms achieve the predicted speed-up in practice. To simulate and compare the algorithms described previously, we consider two scenarios: the SIS inversion problem with $\mathcal{B} = \{0, 1\}$; and the SIS collision problem with $\mathcal{B} = \{0, 1\}$. These experiments simulate the SWIFFT inversion and collision problems, but with smaller parameters.

ISIS inversion $\mathcal{B} = \{0, 1\}$. The parameters we used here are $n = 16, q = 11$ and m ranges from 96 to 176. We compare the extended k-tree algorithm (Minder-Sinclair variant of CPW) with our HNF improvement. We try 5000 instances for each set of parameters starting with different random seeds. Table 1 shows the running-time comparison of algorithms in six sets of parameters. As expected, the problems get easier as the density increases.

Experiment E1 denotes the extended k-tree (CPW) algorithm of Minder and Sinclair (see Section 2.6). Experiment E2 denotes the same algorithm, but with the HNF improvement and using approximate merge (see Section 4.2). Column "max size E" is a theoretical estimate of the maximum number of bits of storage used at any stage of the algorithm during experiment E. The value $\tilde{m}$ in Column "$\tilde{m}$, ℓ_i of E" denotes the actual dimension we used (since for a given dimension m, it is sometimes better to reduce the dimension to get a faster attack).[7] The notation ℓ_i denotes the constraints for each level in the computation; when there are 3 (respectively 4) values ℓ_i listed it means we are performing an 8-list (respectively 16-list) algorithm. Column "time E" denotes the average observed running-time (using a sage implementation run on cores of an Intel 2.7GHz cluster) over 5000 trials for each set of parameters for experiment E.

Table 1: Comparison of algorithms for ISIS inversion $\mathcal{B} = \{0, 1\}$.

Given m	$\tilde{m}$ and ℓ_i **of E1**	$\tilde{m}$ and ℓ_i **of E2**	**max. size E1**	**max. size E2**	**time E1/E2**
96	96, (2, 5, 9)	96, (2, 5, 9)	17.08	14.08	99.87s/8.30s
104	104, (3, 5, 8)	104, (4, 2, 10)	15.62	12.41	30.68s/3.38s
112	112, (4, 4, 8)	112, (4, 4, 8)	14.49	12.00	24.63s/3.39s
128	128, (4, 4, 8)	128, (1, 4, 2, 9)	14.70	11.57	15.78s/2.05s
160	160, (2, 4, 4, 6)	160, (3, 2, 3, 8)	13.08	10.33	8.43s/1.36s
176	176, (3, 3, 4, 6)	160, (3, 2, 3, 8)	12.87	10.33	8.37s/1.36s

The actual running-time follows roughly from the size bound, but not exactly. For instance in algorithm E1, dimension $m = 128$ can be reduced to $\tilde{m} = 112$ which gives a better size bound (from 14.70 to 14.49). However, the actual running-time for keeping $\tilde{m} = 128$ is better than after reducing to 112. To get a more accurate estimate, one can use the bit complexity estimate mentioned in previous sections. To make the comparison fair, we also choose the parameters such that the success probability for our improved algorithm E2 is comparable to that of E1.

Collision on $\mathcal{B} = \{0, 1\}$. We now consider the collision problem for the set $\mathcal{B} = \{0, 1\}$. This simulates the SWIFFT collision problem. Experiment E3 is the Howgrave-Graham-Joux "birthday attack" variant of the Minder-Sinclair CPW algorithm. In other words, we do the CPW algorithm using parameters $\ell_1, \ldots, \ell_t$ and then apply birthday paradox to the final list of entries in $\mathbb{Z}_q^{n-(\ell_1+\cdots+\ell_t)}$. Experiment E4 is the same, but applying the HNF and using approximate merge. The parameters are $n = 16, q = 17$, and m ranges from 96 to 176. The notation used in Table 2 is analogous to that used in Table 1.

[7] When the dimension can be reduced to an instance which has been investigated previously, we do not repeat the experiment but just reproduce the experimental results from the previous instance. e.g. $m = 176$ in experiment E2 can be reduced to the case $\tilde{m} = 160$.

Table 2: Comparison of algorithms for ISIS inversion $\mathcal{B} = \{0, 1\}$.

Given m	$\tilde{m}$ and ℓ_i **of E3**	$\tilde{m}$ and ℓ_i **of E4**	**max. size E3**	**max. size E4**	**time E3/E4**
96	88, (2, 3, 4)	96, (3, 2, 3)	15.39	10.34	23.58s/1.43s
128	128, (1, 3, 2, 4)	96, (3, 2, 3)	14.95	10.34	17.97s/1.43s
144	144, (1, 4, 3, 2)	144, (2, 2, 2, 3)	13.91	10.29	16.21s/1.57s
160	160, (2, 3, 3, 2)	160, (3, 1, 2, 3)	12.85	9.94	8.32s/1.62s
176	176, (3, 2, 3, 2)	176, (1, 1, 2, 2, 3)	12.50	9.59	7.38s/1.46s

7 Conclusions and further work

We have explained how the Hermite normal form reduces the ISIS problem to an "approximate subset-sum" problem, and we have given a variant of the CPW algorithm than can solve such problems. As a result, we have given improved algorithms for inverting and finding collisions for the SWIFFT hash function. Our new methods are approximately 500-1000 times faster than previous methods.

In Section 3 we have analysed the HGJ and BCJ algorithms for ISIS instances of density > 1. Figure 1 illustrates how these algorithms behave as the density grows. While these results are not of interest for the SWIFFT hash function (as it has very high density), they may be relevant to the study of other ISIS problems with small coefficient sets.

Section 4.3 discusses adapting the BCJ algorithm to the case of approximate ISIS. The basic ideas can be applied, but we explain why it seems to be hard to give a general analysis of this case. Nevertheless, we believe these ideas will be of interest to studying ISIS instance with low density and small coefficient set (such as binary LWE and NTRU).

Finally, Section 6 reports on extensive experiments with the CPW algorithm. These results confirm our theoretical analysis, and demonstrate that applying the Hermite normal form to ISIS gives a significant speedup in practice.

There are several questions remaining for future work. One is to determine exact running times for the approximate-BCJ algorithm. Another important challenge is to develop algorithms with lower storage requirements and that can be parallelised or distributed. We note that Pollard-rho-style random walks do not seem to be useful as they lead to running times proportional to $\sqrt{q^n}$, which is usually much worse than the running times considered in this paper.

One final remark: Our general formulation of the HGJ/BCJ/CPW algorithms in terms of taking quotient groups G/H suggests an explanation of why these algorithms cannot be applied to solve the elliptic curve discrete logarithm problem. If $G = E(\mathbb{F}_q)$ is an elliptic curve group of prime order then there are no suitable subgroups H to apply quotients.

References

1. Yuriy Arbitman, Gil Dogon, Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert and Alon Rosen, SWIFFTX: A Proposal for the SHA-3 Standard. Submitted to NIST SHA-3 Competition.
2. Anja Becker, Jean-Sébastien Coron and Antoine Joux, Improved Generic Algorithms for Hard Knapsacks, in K. G. Paterson (ed.), EUROCRYPT 2011, Springer LNCS 6632 (2011) 364–385.
3. Daniel J. Bernstein, Better price-performance ratios for generalized birthday attacks, in Workshop Record of SHARCS07, (2007) `http://cr.yp.to/papers.html#genbday`
4. Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters and Peter Schwabe, FSBday: Implementing Wagner's generalized birthday attack against the SHA-3 round-1 candidate FSB, in B. K. Roy and N. Sendrier (eds.), INDOCRYPT 2009, Springer LNCS 5922 (2009) 18–38.

5. Johannes Buchmann and Richard Lindner, Secure Parameters for SWIFFT, in B. Roy and N. Sendrier (eds.), INDOCRYPT 2009, LNCS 5922 (2009) 1–17.
6. Paul Camion and Jacques Patarin, The Knapsack Hash Function proposed at Crypto'89 can be broken, in D. W. Davies (ed.), EUROCRYPT 1991, Springer LNCS 547 (1991) 39–53.
7. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to algorithms, 2nd ed., MIT press, 2001.
8. 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.
9. Itai Dinur, Orr Dunkelman, Nathan Keller and Adi Shamir, Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems, in R. Safavi-Naini and R. Canetti, CRYPTO 2012, Springer LNCS 7417 (2012) 719–740.
10. Nick Howgrave-Graham, Joseph H. Silverman and William Whyte, A meet-in-the-middle attack on an NTRU private key, Technical Report 004, NTRU Cryptosystems, Jun 2003.
11. Nick Howgrave-Graham and Antoine Joux, New Generic Algorithms for Hard Knapsacks, in H. Gilbert (ed.), EUROCRYPT 2010, Springer LNCS 6110 (2010) 235–256.
12. Nick Howgrave-Graham and Antoine Joux, New Generic Algorithms for Hard Knapsacks (preprint), 17 pages (undated). Available from `www.joux.biz/publications/Knapsacks.pdf`
13. Nick Howgrave-Graham, A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU, in A. Menezes (ed.), CRYPTO 2007, Springer LNCS 4622 (2007) 150–169.
14. Jeffrey C. Lagarias and Andrew M. Odlyzko, Solving low-density subset sum problems, J. ACM, 32(1):229-246, 1985.
15. Vadim Lyubashevsky, On Random High Density Subset Sums, Electronic Colloquium on Computational Complexity (ECCC) 007 (2005)
16. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert and Alon Rosen, SWIFFT: A Modest Proposal for FFT Hashing, in K. Nyberg (ed.), FSE 2008, Springer LNCS 5086 (2008) 54–72.
17. Daniele Micciancio and Chris Peikert, Hardness of SIS and LWE with Small Parameters, in R. Canetti and J. A. Garay (eds.), CRYPTO 2013, Springer LNCS 8042 (2013) 21–39.
18. Daniele Micciancio and Oded Regev, Lattice-based cryptography, in D. J. Bernstein, J. Buchmann and E. Dahmen (eds.), Post Quantum Cryptography, Springer (2009) 147–191.
19. Lorenz Minder and Alistair Sinclair, The Extended k-tree Algorithm, J.Cryptol. 25 (2012) 349–382.
20. Richard Schroeppel and Adi Shamir, A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems, SIAM J. Comput. No. 3 (1981) 456–464.
21. Andrew Shallue, An Improved Multi-set Algorithm for the Dense Subset Sum Problem, in A. J. van der Poorten and A. Stein (eds.), ANTS 2008, Springer LNCS 5011 (2008) 416–429.
22. David Wagner, A Generalized Birthday Problem, in M. Yung (ed.), CRYPTO 2002, Springer LNCS 2442 (2002) 288–303.