Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem

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 $(\mathbf{A}, \mathbf{s})$ to find a short vector $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} \equiv \mathbf{s} \pmod{q}$. We consider algorithms for this problem due to Camion and Patarin; Wagner; Schroeppel and Shamir; Minder and Sinclair; Howgrave-Graham and Joux (HGJ); Becker, Coron and Joux (BCJ). 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; A new method that exploits symmetries to speed up algorithms for Ring-ISIS.

Keywords: Short integer solution problem (SIS), SWIFFT hash function, subset-sum, knapsacks.

1 Introduction

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

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

It is often convenient to write $\mathbf{a} = (a_1, \dots, a_m)$ as a row and $\mathbf{x} = (x_1, \dots, x_m)^T$ as a column so that $s = \mathbf{a}\mathbf{x}$. 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 $s \equiv \mathbf{a}\mathbf{x} \pmod{q}$.

The vector version of this problem is called the *inhomogeneous short 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

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

If we want to be more precise we call this the $(m, n, q, \mathcal{B})$-ISIS problem. The original short 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 $\mathbf{x} \neq \mathbf{0}$. Our algorithms solve both problems.

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 $\mathbf{s} = \mathbf{A}\mathbf{x}$.

We now define the *density* of an ISIS instance, generalising a standard concept in the subset-sum problem. Recall that for integer subset-sum the density is defined to be $m / \log_2(\max\{a_i\})$.

Definition 1. *Let G be a finite group. The* density *of a $(G, m, \mathcal{B})$-ISIS instance is $\delta = \frac{(\#\mathcal{B})^m}{\#G}$.*

If $\delta \leq 1$ then δ is 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 $\mathbf{s} = \mathbf{A}\mathbf{x}$. If $\delta \geq 1$ then δ is the average size of the solution set $\{\mathbf{x} \in \mathcal{B}^m : \mathbf{s} = \mathbf{A}\mathbf{x}\}$, over uniformly chosen elements $(\mathbf{A}, \mathbf{s})$ in $G^m \times G$ such that $\mathbf{s} = \mathbf{A}\mathbf{x}$. If $\delta \ll 1$ then we say the $(G, m, \mathcal{B})$-ISIS problem has low density. If $\delta \approx 1$ then we say we are in the "density 1" case. If $\delta \gg 1$ then we are in the high density case. This informal notion is broadly consistent with the standard notion of density for the subset-sum problem over $\mathbb{Z}$ when $\mathcal{B} = \{0, 1\}$ in the following sense: If one chooses the integers a_i uniformly in an interval $\{0, 1, \ldots, B\} \subset \mathbb{Z}$ then $\max\{a_i\} \approx B$, and the density is approximately $m/\log_2(B)$. Taking $G = \mathbb{Z}_B$, our definition gives density $2^m/B$. One sees that the notion of "low density" with both definitions corresponds to $m < \log_2(B)$, while "high density" is $m > \log_2(B)$ and "density 1" is $m \approx \log_2(B)$.

The ISIS problem has applications in lattice-based cryptography. For example, inverting the SWIFFT hash function of Lyubashevsky, Micciancio, Peikert and Rosen [17] 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 [17] 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 [13] (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 [12]). In fact these arguments are very rough and do not give precise estimates of the actual running time of these attacks (the algorithms sketched in [17] actually require around 2^{148} and 2^{144} bit operations respectively). 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, Minder-Sinclair, Howgrave-Graham and Joux, Becker-Coron-Joux can be used to solve these generalised problems. The four 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 is related to 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 [17]. 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.
4. The SWIFFT hash function and many other cryptographic problems (such as NTRU) are actually based on the Ring-SIS or Ring-ISIS problems (see Section 6). The previous analysis of these algorithms

[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. Their method is to reduce to the sublattices of dimension 206. However the pseudo-collision is not useful to find real collisions for SWIFFT, since in dimension 206 the real collisions for SWIFFT almost certainly do not exist.

has ignored the ring structure. In Section 6 we sketch how to speed up the algorithms by exploiting symmetries. Our main insight is to choose a suitable eigenbasis that allows to include symmetries into our general framework for the algorithms. These ideas do not seem to be compatible with the use of the Hermite normal form, and so do not give further improvements to our attacks on the SWIFFT hash function.

The binary-LWE problem [18] (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.

1.2 Related literature

There is an extensive literature on the approximate subset-sum problem over $\mathbb{Z}$ (given $s \in \mathbb{Z}$ to find $\mathbf{x}$ such that $s \approx \mathbf{ax}$) 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.

Definition 2. *Let G be an abelian group, m an integer and $\mathcal{B}$ a small subset of $\mathbb{Z}$ that contains 0. The $(G, m, \mathcal{B})$-ISIS problem is defined as follows. An instance of the problem is a pair $(\boldsymbol{A}, \boldsymbol{s})$ with $\boldsymbol{A} \in G^m$ and $\boldsymbol{s} \in G$, and a solution is any vector $\boldsymbol{x} \in \mathcal{B}^m$ (if one exists) such that $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{x}$.*

The weight *of a solution $\boldsymbol{x}$ is defined to be $wt(\boldsymbol{x}) = \#\{i : 1 \leq i \leq m, x_i \neq 0\}$. Let $\omega \in \mathbb{N}$. The weight-ω $(G, m, \mathcal{B})$-ISIS problem is: Given $(\boldsymbol{A}, \boldsymbol{s})$, to compute a solution $\boldsymbol{x} \in \mathcal{B}^m$ such that $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{x}$ in G and $wt(\boldsymbol{x}) = \omega$.*

Our three main examples for the group are $G = \mathbb{Z}$, $G = \mathbb{Z}_q$ and $G = \mathbb{Z}_q^n$.

All the algorithms work by reducing to simpler problems (meaning, smaller solution space) of higher density. In our general framework we express this by taking quotients. Indeed, the main conceptual idea of all these algorithms is that high-density instances are easier to solve using brute-force/meet-in-middle ideas, so we always try to reduce the problem to a simpler problem of high density.

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 from $\frac{(\#\mathcal{B})^m}{\#G}$ to $\frac{(\#\mathcal{B})^m}{\#(G/H)}$, 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 $H = \{(0, \dots, 0, g_{\ell+1}, \dots, g_n)^T : g_i \in \mathbb{Z}_q\} \cong \mathbb{Z}_q^{n-\ell}$ so that $G/H \cong \mathbb{Z}_q^\ell$.

High density instances can always be reduced to smaller dimensional instances having density one: Choose a suitable integer ℓ (i.e., so that $(\#\mathcal{B})^{m-\ell} \approx \#G$) 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 $\mathbf{A}'\mathbf{x}' = \mathbf{s}$ in G. Since the number of possible targets remains the same while the number of inputs $\mathbf{x}'$ is reduced to $(\#\mathcal{B})^{m-\ell}$, now the density $\delta = \frac{(\#\mathcal{B})^{m-\ell}}{\#G} \approx 1$. 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 [21] 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 [12]. We briefly recall some details in Section 2.4.

The important paper of Howgrave-Graham and Joux [12] (HGJ) broke 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 [12] presents algorithms for the traditional subset-sum problem, but Section 6 of [12] 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 [12]; we give the details in Section 2.7. Indeed, it is written in [12] 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 later used by Wagner [23] for solving the "k-sum problem". Rather unfairly, these ideas are now often called "Wagner's algorithm", but we will call it CPW and present it in Section 2.5. Minder and Sinclair [20] explained how to use these ideas a bit more effectively (we sketch the details in Section 2.6).

Lyubashevsky [16] noted that the CPW algorithm can be applied to solve high density subset-sum problems. Shallue [22] extended Lyubashevsky's work. Lyubashevsky, Micciancio, Peikert and Rosen [17] explain 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 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 $\#(G/(H_i \cap H_j)) = \#(G/H_i) \cdot \#(G/H_j)$. All algorithms 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).

We consider one step of the merge algorithm[5]. Let $H^\flat, H, H^\sharp$ be subgroups of G that denote subgroups used in the CPW/HGJ/BCJ algorithms. We are merging modulo H a pair of lists L_1 and L_2 that are "partial solutions" modulo $H^\flat$. In other words, the output is a set of solutions to the problem $\mathbf{A}\mathbf{x} \equiv \mathbf{s} \pmod{H \cap H^\flat}$ for $\mathbf{x} \in \mathcal{X}$. For future processing, the output includes information about $\mathbf{A}\mathbf{x} \pmod{H^\sharp}$. The details are given as Algorithm 1.

The running time of the algorithm depends on the cost of sorting L_2 and searching $\mathbf{v}$ in L_2 for every $\mathbf{u}$ in L_1, which is $O(\#L_2 \log_2(\#L_2) + \#L_1 \log_2(\#L_2))$ i.e., $\tilde{O}(\max(\#L_1, \#L_2))$. 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 $\mathbf{u}$ give rise to matches between the two lists L_1 and L_2. Treating the function from $\mathcal{X}$ to G/H given by $\mathbf{x} \mapsto \mathbf{A}\mathbf{x} \pmod{H}$ as pseudorandom, the total number of $(\mathbf{x}_1, \mathbf{x}_2)$ pairs can be bounded by

[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 name used by several previous authors so we continue to use it.

Algorithm 1 Basic merge algorithm

```
INPUT: $L_1 = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H}) : \mathbf{A}\mathbf{x} \equiv R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_1\}$,
       $L_2 = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H}) : \mathbf{A}\mathbf{x} \equiv \mathbf{s} - R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_2\}$
OUTPUT: $L = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H^\sharp}) : \mathbf{A}\mathbf{x} \equiv \mathbf{s} \pmod{H \cap H^\flat}, \mathbf{x} \in \mathcal{X}\}$
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 $L$
```

$\#L_1 \cdot \#L_2/\#(G/H)$. Hence, the heuristic running time is $\tilde{O}(\max\{\#L_1, \#L_2, \#L_1\#L_2/\#(G/H)\})$. (This analysis includes the correction by May and Meurer to the analysis in [12], 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 [21] 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 $(\#\mathcal{B})^{m/2}$ 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)

The CPW 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 [19] notes that the algorithm can be used to solve (I)SIS. We will explain that this algorithm also can be used to solve the $(G, m, \mathcal{B})$-ISIS problem.

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 $G/(H_{i_1} \cap H_{i_2}) \cong (G/H_{i_1}) \oplus (G/H_{i_2})$ for any $1 \le i_1 < i_2 \le t$ 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 $1 \le i < t$ and $\#(G/H_t) \approx (\#G)^{2/(t+1)}$, although Minder and Sinclair [20] obtain improvements by relaxing these conditions.

For the (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 $H_1 = \{(0, \ldots, 0, g_{\ell+1}, \ldots, g_n)^T : g_i \in \mathbb{Z}_q\}$ 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 $H_2 = \{(g_1, \dots, g_\ell, 0, \dots, 0, g_{2\ell+1}, \dots, g_n)^T : g_i \in \mathbb{Z}_q\}$). 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 $\mathcal{X}_1 = \{(x_1, \dots, x_u, 0, \dots, 0) \in \mathcal{B}^m\}$ and

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

for $2 \leq j \leq k$.

Level 0: The CPW algorithm works by first constructing $k = 2^t$ lists $L_j^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H_1}) : \mathbf{x} \in \mathcal{X}_j\}$ for $1 \leq j \leq k-1$ and $L_k^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x} - \mathbf{s} \pmod{H_1}) : \mathbf{x} \in \mathcal{X}_k\}$. 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.)

Level 1: Use Algorithm 1 to merge the lists from level 0 to compute the $k/2$ new lists $L_1^{(1)}, \dots, L_{k/2}^{(1)}$, where for $1 \leq j \leq k/2 - 1$ each $L_j^{(1)}$ contains pairs $(\mathbf{x}_1, \mathbf{x}_2) \in L_{2j-1}^{(0)} \times L_{2j}^{(0)}$ such that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \equiv \mathbf{0} \pmod{H_1}$ and $L_{k/2}^{(1)}$ contains pairs $(\mathbf{x}_1, \mathbf{x}_2) \in L_{k-1}^{(0)} \times L_k^{(0)}$ such that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) - \mathbf{s} \equiv \mathbf{0} \pmod{H_1}$. In other words, the new lists $L_j^{(1)}$ for $1 \leq j \leq k/2$ contain elements $\mathbf{x}_1 + \mathbf{x}_2$ that are "correct" for the quotient G/H_1. To optimise the running time one only computes $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_2}$ at this level; the merge can be performed efficiently using Algorithm 1. The output of the algorithm is $k/2$ new lists, for $1 \leq j \leq k/2 - 1$, $L_j^{(1)} = \{(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_2}) : \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \equiv \mathbf{0} \pmod{H_1}, \mathbf{x}_1 \in L_{2j-1}^{(0)}, \mathbf{x}_2 \in L_{2j}^{(0)}\}$, and $L_{k/2}^{(1)} = \{(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_2}) : \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) - \mathbf{s} \equiv \mathbf{0} \pmod{H_1}, \mathbf{x}_1 \in L_{k-1}^{(0)}, \mathbf{x}_2 \in L_k^{(0)}\}$.

Level $i \geq 2$: Use Algorithm 1 to merge the lists $L_{2j-1}^{(i-1)}$ and $L_{2j}^{(i-1)}$ from level $i-1$. The output of the algorithm is $k/2^i$ lists $L_j^{(i)}$ containing elements that are "correct" for the quotient $G/(H_1 \cap \cdots \cap H_i)$. Precisely, for $1 \leq j \leq k/2^i - 1$, $L_j^{(i)} = \{(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_{i+1}}) : \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \equiv \mathbf{0} \pmod{H_i}, \mathbf{x}_1 \in L_{2j-1}^{(i-1)}, \mathbf{x}_2 \in L_{2j}^{(i-1)}\}$ and $L_{k/2^i}^{(i)} = \{(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_{i+1}}) : \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) - \mathbf{s} \equiv \mathbf{0} \pmod{H_i}, \mathbf{x}_1 \in L_{k/2^{i-1}-1}^{(i-1)}, \mathbf{x}_2 \in L_{k/2^{i-1}}^{(i-1)}\}$.

Level t: Merge the two lists $L_1^{(t-1)}$ and $L_2^{(t-1)}$ to get one list $L_1^{(t)}$ by ensuring the solutions are correct modulo H_t. In other words the list contains elements that are "correct" for $G/(H_1 \cap \cdots \cap H_t) = G$. The output of Algorithm 1 at this stage is the list $L_1^{(t)} = \{\mathbf{x}_1 + \mathbf{x}_2 : \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) - \mathbf{s} \equiv \mathbf{0} \pmod{H_t}, \mathbf{x}_1 \in L_1^{(t-1)}, \mathbf{x}_2 \in L_2^{(t-1)}\}$.

Success Probability. The heuristic analysis of the success probability is as follows.

If $L_1^{(t)}$ is not empty, the CPW algorithm succeeds to find a solution for (I)SIS. The expected size of the lists on each level is:

$$\#L^{(0)} \approx (\#\mathcal{B})^{m/2^t},$$

$$\#L^{(i)} \approx \#L^{(i-1)} \#L^{(i-1)}/q^\ell, 1 \leq i \leq t-1,$$

$$\#L^{(t)} \approx \#L^{(t-1)} \#L^{(t-1)}/q^{2\ell}.$$

To keep $\#L_1^{(t)} \approx 1$, the standard argument is that we want the lists $L^{(1)}, \dots, L^{(t-1)}$ to all be roughly the same size. It follows that we desire $\ell \approx n/(t+1)$, $(\#\mathcal{B})^{2m/k}/(\#G)^{1/(t+1)} \approx (\#\mathcal{B})^{m/k}$ and so $(\#G)^{1/(t+1)} \approx (\#\mathcal{B})^{m/k}$ (i.e., $2^t/(t+1) \approx \log_2((\#\mathcal{B})^m)/\log_2(\#G)$). Then the final list at level t has expected size ≈ 1. We refer to [6, 23, 19, 17, 20] for full details and heuristic analysis. The time complexity of CPW is $\tilde{O}(k \cdot (\#\mathcal{B})^{m/k}) = \tilde{O}(2^t \cdot (\#\mathcal{B})^{m/2^t})$. Lyubashevsky [16] and Minder and Sinclair [20] provide some rigorous analysis of success probability of the CPW algorithm that supports the validity of the heuristic analysis.

Running Time. In practice for a given (I)SIS instance, one takes $k = 2^t$ to be as large as possible subject to the constraint $(\#\mathcal{B})^{m/2^t} \geq (\#G)^{1/(t+1)}$, in other words t is the largest integer such that $2^t/(t+1) \leq \log_2((\#\mathcal{B})^m)/\log_2(\#G)$ (recall that the density was defined to be $(\#\mathcal{B})^m/(\#G)$). Hence the size of k is governed by the density of the instance (higher density means larger k). When the density is 1 (i.e., $(\#\mathcal{B})^m \approx (\#G)$) 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.

Assume that for some integer t, the density for the (I)SIS instance (we have $G = \mathbb{Z}_q^n$) satisfies the constraint:

$$\frac{2^{t-1}}{t} < \frac{\log_2((\#\mathcal{B})^m)}{\log_2(q^n)} < \frac{2^t}{t+1},$$

since $\frac{\log_2((\#\mathcal{B})^m)}{\log_2(q^n)} < \frac{2^t}{t+1}$ the largest k we can choose is 2^{t-1}. Directly using the CPW algorithm the time complexity is $\tilde{O}(2^t \cdot (\#\mathcal{B})^{m/2^{t-1}})$. However, since $\frac{2^{t-1}}{t} < \frac{\log_2((\#\mathcal{B})^m)}{\log_2(q^n)}$, the density is higher than what CPW using 2^{t-1} lists needs to find a solution. Hence, one can reduce the density: Choose an integer ℓ_0 such that $\frac{\log_2((\#\mathcal{B})^{m-\ell_0})}{\log_2(q^n)} \approx \frac{2^{t-1}}{t}$. In other words, $(\#\mathcal{B})^{(m-\ell_0)/2^{t-1}} \approx q^{n/t}$. Then set ℓ_0 entries of $\mathbf{x}$ to be zero. In other words, delete the corresponding ℓ_0 columns of $\mathbf{A}$ to get an $n \times (m - \ell_0)$ matrix $\mathbf{A}'$ and let $\mathbf{x}'$ be the corresponding vector in $\mathbb{Z}^{m-\ell_0}$. One can use the CPW algorithm with $k = 2^{t-1}$ to solve $\mathbf{A}'\mathbf{x}' = \mathbf{s}$ in G. The time complexity is reduced to $\tilde{O}(2^t \cdot (\#\mathcal{B})^{(m-\ell_0)/2^{t-1}}) = \tilde{O}(2^t \cdot q^{n/t})$.

Remarks. 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 the algorithm is completely deterministic, and so always gives the same solution set, but to obtain different solutions one can apply a random permutation to the problem before running the algorithm.

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 [16]). We also mention the work of Shallue [22], which gives a rigorous analysis of the CPW algorithm for the modular subset-sum problem.

2.6 Minder and Sinclair refinement of CPW

In this section, we introduce the work of Minder and Sinclair [20] that allows a finer balancing of parameters. This allows the CPW algorithm to be used for larger values of k than the density of the (I)SIS instance might predict. Assume the density for a (I)SIS instance ($G = \mathbb{Z}_q^n$) satisfies the constraint

$$\frac{2^{t-1}}{t} < \frac{\log_2((\#\mathcal{B})^m)}{\log_2(q^n)} < \frac{2^t}{t+1} \tag{3}$$

for some integer t.

Instead of reducing the density as described in the previous section, Minder and Sinclair proposed the "extended k-tree" algorithm to make use of the extra density. When the density satisfies (3), Minder and Sinclair use $k = 2^t$ (In the previous section, the CPW algorithm chooses $k = 2^{t-1}$) by choosing appropriate subgroups H_i. Let $\ell_i \geq 1$ be chosen later, subject to $\ell_1 + \ell_2 + \cdots + \ell_t = n$. The subgroup $H_1 = \{(0, \cdots, 0, g_{\ell_1+1}, \cdots, g_n)^T : g_i \in \mathbb{Z}_q\}$ such that $G/H_1 \cong \mathbb{Z}_q^{\ell_1}$ corresponds to the first ℓ_1 positions of the vector. Similarly, $H_2 = \{(g_1, \cdots, g_{\ell_1}, 0, \cdots, 0, g_{\ell_1+\ell_2+1}, \cdots, g_n) : g_i \in \mathbb{Z}_q\}$ such that $G/H_2 \cong \mathbb{Z}_q^{\ell_2}$ corresponds to the next ℓ_2 positions of the vector. Finally, $H_t = \{(g_1, \cdots, g_{\ell_1+\cdots+\ell_{t-1}}, 0, \cdots, 0) : g_i \in \mathbb{Z}_q\}$ corresponds to the last ℓ_t positions of the vector. Denote by $L^{(i)}$ any of the lists at the i-th stage of the algorithm. The time complexity for Minder and Sinclair's algorithm is $\tilde{O}(2^t \cdot \max_{0\leq i\leq t}(\#L^{(i)}))$. To minimise the running time, one needs to minimise $\max_{0\leq i\leq t}(\#L^{(i)})$. The expected size of the lists on each level is

$$\#L^{(0)} = (\#\mathcal{B})^{m/2^t},$$

$$\#L^{(i)} = \#L^{(i-1)}\#L^{(i-1)}/q^{\ell_i}, 1 \le i \le t.$$

Write $\#L^{(i)} = 2^{b_i}$ where $b_0 = m\log_2(\#\mathcal{B})/2^t$, $b_i = 2b_{i-1} - \log_2(q)\ell_i$. To minimize the time complexity, one computes the optimal values ℓ_i by solving the following integer program.

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

Theorem 3.1 in [20] shows that the solution to the above linear program (i.e., removing the constraint $\ell_i \in \mathbb{Z}$) is $\ell_2 = \cdots = \ell_{t-1} = (n-\ell_1)/t$ and $\ell_t = 2(n-\ell_1)/t$ where ℓ_1 satisfies $(\#\mathcal{B})^{m/2^t} \cdot (\#\mathcal{B})^{m/2^t}/q^{\ell_1} = q^{(n-\ell_1)/t}$. This gives $\max_{0\le i\le t}(\#L^{(i)}) = q^{(n-\ell_1)/t}$. From (3), we have $0 < \ell_1 < n/(t+1)$. The time complexity of Minder and Sinclair's algorithm is $\tilde{O}(2^t \cdot \max_{0\le i\le t}(\#L^{(i)})) = \tilde{O}(2^t \cdot q^{(n-\ell_1)/t})$.[6] It follows that

$$(\#\mathcal{B})^{m/2^t} < \max_{0\le i\le t}(\#L^{(i)}) = q^{(n-\ell_1)/t} < (\#\mathcal{B})^{m/2^{t-1}}.$$

At a high level, Minder and Sinclair make use of the extra density in the instance to add one more level that eliminates ℓ_1 coordinates. The time complexity for Minder and Sinclair's refinement of CPW is better than the original version described in the previous section since $\tilde{O}(2^t \cdot q^{(n-\ell_1)/t}) < \tilde{O}(2^t \cdot (\#\mathcal{B})^{m/2^{t-1}})$ and $\tilde{O}(2^t \cdot q^{(n-\ell_1)/t}) < \tilde{O}(2^t \cdot q^{n/t})$.

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

We now present the HGJ algorithm, that can be applied even for instances of the $(G, m, \mathcal{B})$-ISIS problem of density ≤ 1. The algorithm 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 [12] 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 [12] 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 [12] 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 $\mathbf{s} = \mathbf{A}\mathbf{x}$ in G where $\mathbf{x} \in \mathcal{B}^m$ has weight $\mathrm{wt}(\mathbf{x}) = \omega$. Our goal is to compute $\mathbf{x}$. 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

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

[6] In practice, we want ℓ_i to be integers, good parameter choices can be obtained by using $\lceil \ell_i \rceil$, see [20] for details. The time complexity can be a little bit worse, however we believe this rounding does not affect the asymptotic complexity.

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 [12] gives several solutions to the problem of merging lists, including a 4-list merge. But the main algorithm in [12] 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.

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 inconvenience is that we may not know exactly 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.

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 [12], "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 similar form to those in Section 2.5, but we now require $H_1 \cap H_2 \cap H_3 \neq \{0\}$) 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 $H_2 \cap H_3$ etc.

In [12], 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 [12], 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 $\binom{\omega}{\omega/2}$ 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 to solve the (I)SIS problem when $\omega = m/2$ (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 $\mathcal{X} \subset \mathcal{B}^m$ 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, $\mathcal{B} = \{0,1\}$, $G = \mathbb{Z}_q^n$, and where we solve ISIS instances of density 1 (so that $2^m \approx q^n$) with a solution of weight $m/2$. 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 $2^{H(a,b)\cdot m}$ where $H(x,y) = -x\log_2(x) - y\log_2(y) - (1-x-y)\log_2(1-x-y)$.

Choose subgroups H_1, H_2, H_3 (of the similar form to Section 2.6) such that $\#(G/H_i) = q^{\ell_i}$. Fix $\alpha = 0.0267, \beta = 0.0168$ and $\gamma = 0.0029$ and also integers ℓ_1, ℓ_2, ℓ_3 such that $q^{\ell_1} \approx q^{0.2673n} = 2^{0.2673m}, q^{\ell_2} \approx 2^{0.2904m}$ and $q^{\ell_3} \approx 2^{0.2408m}$. Note that $\#(G/(H_1 \cap H_2 \cap H_3)) \approx q^{0.2015n} \approx 2^{0.2015m}$.

Theorem 1. *(Becker-Coron-Joux) With notation as above, and assuming heuristics about the pseudorandomness of $\boldsymbol{Ax}$, the BCJ algorithm runs in time $\tilde{O}(2^{0.2912m})$.*

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 $L_1^{(1)} = \{(\mathbf{x}, \mathbf{Ax}) : \mathbf{x} \in \mathcal{X}_1, \mathbf{Ax} \equiv R_1 \pmod{H_1 \cap H_2 \cap H_3}\}$ and $L_2^{(1)}$, which is the same except $\mathbf{Ax} \equiv \mathbf{s} - R_1 \pmod{H_1 \cap H_2 \cap H_3}$. The expected size of the lists is $2^{H(1/4+\alpha,\alpha)m}/q^{\ell_1+\ell_2+\ell_3} = 2^{0.2173m}$ and merging requires $\tilde{O}((2^{0.2173m})^2/q^{n-\ell_1-\ell_2-\ell_3}) = \tilde{O}(2^{0.2331m})$ 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 $L_1^{(2)} = \{(\mathbf{x}, \mathbf{Ax}) : \mathbf{x} \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}, \mathbf{Ax} \equiv R_2 \pmod{H_2 \cap H_3}\}$ 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 $2^{H(1/8+\alpha/2+\beta,\alpha/2+\beta)m}/q^{\ell_2+\ell_3} = 2^{0.2790m}$ and the cost to merge is $\tilde{O}(2^{2\cdot 0.2790m}/q^{\ell_1}) = \tilde{O}(2^{(2\cdot 0.2790-0.2673)m}) = \tilde{O}(2^{0.2907m})$.

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

$$2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m}/q^{\ell_3} \approx 2^{0.2908m}$$

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

$$\tilde{O}(\sqrt{2^{H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)m}}) = \tilde{O}(2^{0.2658m}).$$

Merging the lists takes $\tilde{O}(2^{2\cdot 0.2908}/q^{\ell_2}) = \tilde{O}(2^{0.2912m})$ time. Thus the time complexity of the BCJ algorithm is $\tilde{O}(2^{0.2912m})$. □

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. We call such splittings "valid". Hence, the number of ways to split $\mathbf{x}$ in this way is

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

For randomly chosen $R_1 \in G/(H_1 \cap H_2 \cap H_3)$, there is a good chance that a valid splitting exists if $\mathcal{N}_1 \geq q^{\ell_1+\ell_2+\ell_3}$. Indeed, the expected number of valid splittings should be roughly $\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3}$. Hence, we choose $\mathcal{N}_1 \approx q^{\ell_1+\ell_2+\ell_3}$ to make sure a valid splitting exists at this stage with significant probability.

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

$$\mathcal{N}_2 = \binom{(1/4+\alpha)m}{(1/8+\alpha/2)m}\binom{\alpha m}{\alpha m/2}\binom{(3/4-2\alpha)m}{\beta m, \beta m, (3/4-2\alpha-2\beta)m}.$$

The expected number of valid splittings at this stage should be roughly $(\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2$. For randomly chosen $R_2 \in G/(H_2 \cap H_3)$, there is a good chance to have a valid splitting if $(\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2 \geq 1$ (remember that this stage requires splitting two solutions from the first stage). Hence, we choose $\mathcal{N}_1 \approx q^{\ell_1+\ell_2+\ell_3}, \mathcal{N}_2 \approx q^{\ell_2+\ell_3}$.

In the final stage (again assuming good splittings in the second stage), the number of ways to split is

$$\mathcal{N}_3 = \binom{(1/8+\alpha/2+\beta)m}{(1/16+\alpha/4+\beta/2)m}\binom{(\beta+\alpha/2)m}{(\beta/2+\alpha/4)m}\binom{(7/8-\alpha-2\beta)m}{\gamma m, \gamma m, (7/8-\alpha-2\beta-2\gamma)m}.$$

The expected number of valid splittings is $(\mathcal{N}_1/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2/q^{\ell_2+\ell_3})^2(\mathcal{N}_3/q^{\ell_3})^4$, which we require to be ≥ 1. Hence, we choose $\mathcal{N}_1 \approx q^{\ell_1+\ell_2+\ell_3}, \mathcal{N}_2 \approx q^{\ell_2+\ell_3}, \mathcal{N}_3 \approx q^{\ell_3}$. 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 significantly larger than 0. 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

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

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

2.9 Summary

Despite the large literature on the topic, summarised above, one sees there are only two fundamental ideas that are used by all these 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 [12, 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 to compute 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 BCJ algorithms can be improved when one is solving instances of density > 1. This was anticipated in [13]: "further improvements can be obtained if, in addition, we seek one solution among many". We now give a very approximate heuristic analysis of this situation. We focus on the case $\mathcal{B} = \{0,1\}^m$ and $G = \mathbb{Z}_q^n$. As with previous work, we are not able to give general formulae for the running time as a function of the density, as the parameters in the algorithms depend in subtle ways on each other. Instead, we fix some reference instances, compute optimal parameters for them, and give the running times.

Let the number of solutions to the original (I)SIS instance be $\mathcal{N}_{sol} \geq 1$. We consider at most t levels of recursion. The subgroups $H_1, H_2, \cdots, H_t$, where $\#(G/H_i) = q^{\ell_i}$, 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 $1 \le i \le t$. The success probability is then justified by requiring

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

for all $1 \le i \le t$. We now assume a best-case scenario, that all the splittings of all the $\mathcal{N}_{sol} \ge 1$ solutions are distinct. This assumption 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. We address this assumption in Section 3.1. Then the success condition changes, for all $1 \le i \le t$, to

$$\mathcal{N}_{sol} \frac{\mathcal{N}_1}{q^{\ell_1+\cdots+\ell_t}} \left(\frac{\mathcal{N}_2}{q^{\ell_2+\cdots+\ell_t}}\right)^2 \cdots \left(\frac{\mathcal{N}_i}{q^{\ell_i+\cdots+\ell_t}}\right)^{2^{i-1}} \ge 1. \tag{5}$$

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

$$\begin{aligned}
\text{minimize } & T = \max\left(\frac{\#(L^{(1)})^2}{q^{n-\ell_1-\ell_2-\ell_3}}, \frac{\#(L^{(2)})^2}{q^{\ell_1}}, \frac{\#(L^{(3)})^2}{q^{\ell_2}}, \sqrt{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}}\right) \\
\text{subject to } & \#L^{(1)} = \frac{\#\mathcal{X}_{1/4+\alpha,\alpha}}{q^{\ell_1+\ell_2+\ell_3}}, \\
& \#L^{(2)} = \frac{\#\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}}{q^{\ell_2+\ell_3}}, \\
& \#L^{(3)} = \frac{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}}{q^{\ell_3}}, \\
& \text{equation (5) holds for all } 1 \le i \le 3 \\
& \ell_i \in \mathbb{N}, 1 \le i \le 3 \\
& \alpha, \beta, \gamma \in \mathbb{R}_{\ge 0}.
\end{aligned}$$

For the ISIS problem $\mathcal{B} = \{0,1\}^m$ given q and n, the density is $2^m/q^n$. For a given density $2^{c_1 m}$ one can estimate the time complexity as $\tilde{O}(2^{c_2 m})$ by solving the above linear program and choosing the optimal parameters α, β, γ and ℓ_1, ℓ_2, ℓ_3. We use LINGO 11.0 to do the optimization and obtain Table 2 and 3 giving calculations for HGJ/BCJ. For comparison we recall in Table 1 the results of Sections 2.5 and 2.6 on the time complexity of CPW. We draw Figure 1, which indicates how the density affects the asymptotic complexity for CPW[7], HGJ and BCJ.

Table 1. Time complexity of CPW for different density.

t	$2^{c_1 m}$	$\tilde{O}(2^{c_2 m})$
1	1	$\tilde{O}(2^{0.5m})$
2	$2^{0.25m}$	$\tilde{O}(2^{0.25m})$
3	$2^{0.5m}$	$\tilde{O}(2^{0.125m})$
4	$2^{0.6875m}$	$\tilde{O}(2^{0.0625m})$

[7] We remark that [10] contains a similar figure regarding the complexity of the CPW algorithm.

Table 2. Time complexity of HGJ for different density.

t	$2^{c_1 m}$	$\tilde{O}(2^{c_2 m})$	q^{ℓ_1}	q^{ℓ_2}	q^{ℓ_3}
2	$2^0 = 1$	$\tilde{O}(2^{0.3371m})$	$2^{0.25m}$	$2^{0.25m}$	–
2	$2^{0.025m}$	$\tilde{O}(2^{0.3121m})$	$2^{0.275m}$	$2^{0.25m}$	–
2	$2^{0.045m}$	$\tilde{O}(2^{0.2928m})$	$2^{0.2928m}$	$2^{0.2507m}$	–
2	$2^{0.055m}$	$\tilde{O}(2^{0.2878m})$	$2^{0.2878m}$	$2^{0.2557m}$	–
2	$2^{0.085m}$	$\tilde{O}(2^{0.2728m})$	$2^{0.2728m}$	$2^{0.2707m}$	–
3	$2^{0.15m}$	$\tilde{O}(2^{0.2535m})$	$2^{0.2617m}$	$2^{0.1711m}$	$2^{0.125m}$
3	$2^{0.2m}$	$\tilde{O}(2^{0.2368m})$	$2^{0.2617m}$	$2^{0.1878m}$	$2^{0.125m}$
3	$2^{0.3m}$	$\tilde{O}(2^{0.2070m})$	$2^{0.2670m}$	$2^{0.2070m}$	$2^{0.1303m}$
3	$2^{0.5m}$	$\tilde{O}(2^{0.1887m})$	$2^{0.2677m}$	$2^{0.1879m}$	$2^{0.1669m}$

Table 3. Time complexity of BCJ for different density.

t	$2^{c_1 m}$	$\tilde{O}(2^{c_2 m})$	q^{ℓ_1}	q^{ℓ_2}	q^{ℓ_3}	α	β	γ
3	$2^0 = 1$	$\tilde{O}(2^{0.2912m})$	$2^{0.2673m}$	$2^{0.2904m}$	$2^{0.2408m}$	0.0267	0.0168	0.0029
3	$2^{0.025m}$	$\tilde{O}(2^{0.2829m})$	$2^{0.2463m}$	$2^{0.2802m}$	$2^{0.2829m}$	0.02578	0.01973	0.00534
3	$2^{0.045m}$	$\tilde{O}(2^{0.2794m})$	$2^{0.2604m}$	$2^{0.2794m}$	$2^{0.2770m}$	0.02818	0.01833	0.00453
3	$2^{0.055m}$	$\tilde{O}(2^{0.2765m})$	$2^{0.2634m}$	$2^{0.2765m}$	$2^{0.2649m}$	0.02651	0.01705	0.00391
3	$2^{0.085m}$	$\tilde{O}(2^{0.2579m})$	$2^{0.2404m}$	$2^{0.2564m}$	$2^{0.1818m}$	0.01082	0.00888	0.00131
3	$2^{0.15m}$	$\tilde{O}(2^{0.2499m})$	$2^{0.2430m}$	$2^{0.2499m}$	$2^{0.1554m}$	0.00102	0.00971	0.00023
3	$2^{0.2m}$	$\tilde{O}(2^{0.2357m})$	$2^{0.2357m}$	$2^{0.2358m}$	$2^{0.2036m}$	0.00231	0.01171	0.00206
3	$2^{0.3m}$	$\tilde{O}(2^{0.2070m})$	$2^{0.2670m}$	$2^{0.2070m}$	$2^{0.1303m}$	0	0	0
3	$2^{0.5m}$	$\tilde{O}(2^{0.1887m})$	$2^{0.2677m}$	$2^{0.1879m}$	$2^{0.1669m}$	0	0	0

These results demonstrate that, although CPW is the best choice for very high density instances, the HGJ/BCJ algorithms do perform better when the density is increased and they are better than CPW for quite a large range of densities. The final rows of Table 3 show that the BCJ algorithm becomes exactly the HGJ algorithm once the density is sufficiently high.

To invert the SWIFFT hash function the parameters are $\mathcal{B} = \{0,1\}^m$, $G = \mathbb{Z}_q^n$, $m = 1024$, $q = 257$, $n = 64$ and so the density is $2^{0.5m}$. Figure 1 therefore confirms that, for this problem, the CPW algorithm is the right choice.

3.1 Heuristic Justification of Assumption

The above analysis is based on the strong simplifying assumption that all the splittings of all the $\mathcal{N}_{sol} = 2^{c_1 m}$ solutions are distinct, and it should not be assumed that the HGJ and BCJ algorithms perform exactly this well. However, we believe the assumption is reasonable when the density is moderate. The reason is given as follows.

We have $\mathcal{N}_{sol} = 2^{c_1 m}$ solutions $\mathbf{x} \in \mathcal{B}^m$ such that $\mathbf{A}\mathbf{x} = \mathbf{s}$. We suppose all these solutions look like independently chosen binary strings of Hamming weight very close to $m/2$. Consider one solution $\mathbf{x}$. In the first level of the recursion we split $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_{1/4+\alpha,\alpha}$. There are $\mathcal{N}_1$ ways to do this, where $\mathcal{N}_1$ is given in equation (4) and all these splittings have, by definition, distinct values for $\mathbf{x}_1$. Turning this around, the probability that a vector $\mathbf{x}_1 \in \mathcal{X}_{1/4+\alpha,\alpha}$ appears as a splitting of $\mathbf{x}$ should be $p_1 = \frac{\mathcal{N}_1}{\#\mathcal{X}_{1/4+\alpha,\alpha}}$. Now consider splitting a second solution $\mathbf{x}' = \mathbf{x}'_1 + \mathbf{x}'_2$. The probability that a value $\mathbf{x}'_1$ is equal to one of

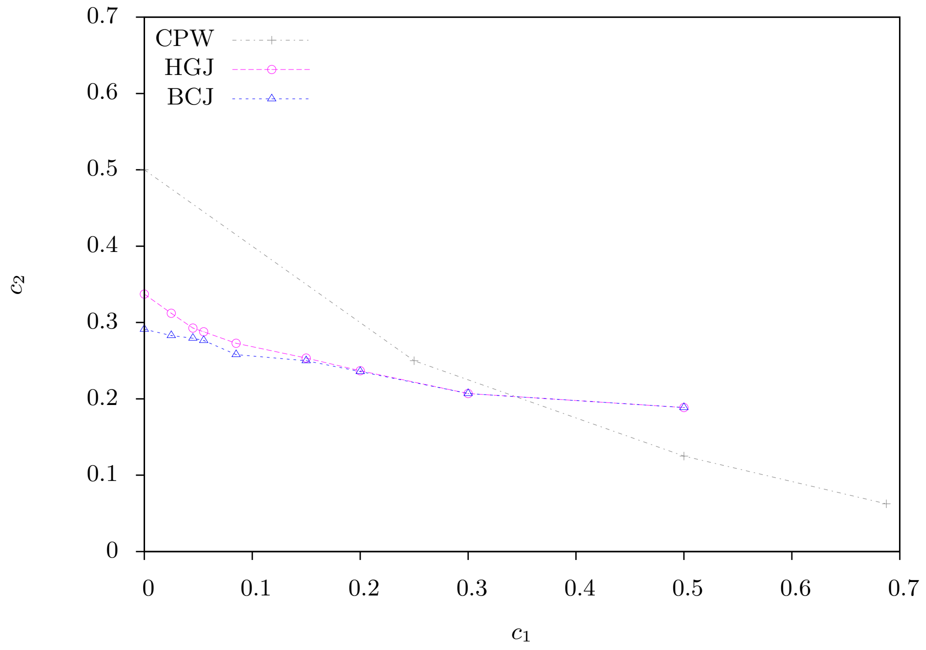

Fig. 1. Heuristic comparison of the performance of CPW, HGJ and BCJ algorithms on ISIS instances of density ≥ 1. The horizontal axis is the value c_1 such that the density (expected number of solutions) is $2^{c_1 m}$. The vertical axis is the constant c_2 such that the heuristic asymptotic complexity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

the previous values $\mathbf{x}_1$ is p_1. Hence, the total number of "new" solutions $\{\mathbf{x}_1, \mathbf{x}'_1\}$ is $\mathcal{N}_1 + (1-p_1)\mathcal{N}_1$. The following Lemma extends this argument.

Lemma 1. *Let X be a set. Suppose distinct subsets X_i of X of size $\mathcal{N}_1$ are chosen uniformly at random for $1 \leq i \leq t$. Let $p = \mathcal{N}_1/\#X$. Then the expected size of $\cup_{i=1}^{t} X_i$ is $(1-(1-p)^t)\mathcal{N}_1/p$.*

Proof. The probability that any given $\mathbf{x} \in X_1$ lies in X_2 is p, so the expected size of $X_1 \cap X_2$ is $p\#X_1$. So we expect $\#(X_1 \cup X_2) = (2-p_1)\mathcal{N}_1 = (1+(1-p))\mathcal{N}_1$.

The probability that any $\mathbf{x} \in X_1 \cup X_2$ lies in X_3 is p, so the expected size of $(X_1 \cup X_2) \cap X_3 = p\#(X_1 \cup X_2) = p(1+(1-p))$. Hence, we expect $\#(X_1 \cup X_2 \cup X_3) = \#(X_1 \cup X_2) + \#X_3 - \#((X_1 \cup X_2) \cap X_3) = (1+(1-p)+1-p(1+(1-p)))\mathcal{N}_1 = (1+(1-p)+(1-p)^2)\mathcal{N}_1$. More generally, one can show by induction that the expected value of $\#(X_1 \cup \cdots \cup X_t)$ is $(1+(1-p)+\cdots+(1-p)^{t-1})\mathcal{N}_1 = \frac{1-(1-p)^t}{p}\mathcal{N}_1$. $\square$

Lemma 1 indicates that when we split all $\mathcal{N}_{sol}$ solutions $\mathbf{x}$, the total number of vectors $\mathbf{x}_1 \in \mathcal{X}_{1/4+\alpha,\alpha}$ should be roughly $(1-(1-p_1)^{\mathcal{N}_{sol}})\mathcal{N}_1/p_1$. If $p_1\mathcal{N}_{sol}$ is very small ($\ll 1$) then one has $(1-p_1)^{\mathcal{N}_{sol}} \approx 1 - p_1\mathcal{N}_{sol}$ and so $(1-(1-p_1)^{\mathcal{N}_{sol}})/p_1 \approx \mathcal{N}_{sol}$. In other words, almost all the splittings of all the $\mathcal{N}_{sol}$ solutions are distinct. For the values of α listed in Tables 2 and 3, $\max(p_1) = 2^{-0.2143m}$, and so if $\mathcal{N}_{sol} \leq 2^{0.2m}$ then $p_1\mathcal{N}_{sol} \leq 2^{-0.0143m}$. Hence, the assumption made earlier seems justified when the density is less that $2^{0.2m}$. We now assume this is the case.

In the second level of the recursion we have $\mathcal{N}'_{sol} = \mathcal{N}_{sol}\frac{\mathcal{N}_1}{q^{\ell_1+\ell_2+\ell_3}}$ solutions. Each solution has $\mathcal{N}_2$ splittings as a sum of vectors in $\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$. Let $p_2 = \frac{\mathcal{N}_2}{\#\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}}$ be the probability a random vector appears as such a splitting. Lemma 1 shows that, as long as $p_2\mathcal{N}'_{sol} \ll 1$, we again expect almost all the splittings to be distinct.

For all values α, β listed in Tables 2 and 3 we have $\max(p_2) = 2^{-0.2469m}$. Since $\mathcal{N}_{sol}\frac{\mathcal{N}_1}{q^{\ell_1+\ell_2+\ell_3}} \leq \mathcal{N}_{sol} \leq 2^{0.2m}$ (when $\mathcal{N}_{sol} = 1$, one chooses $q^{\ell_1+\ell_2+\ell_3} \approx \mathcal{N}_1$, and if $\mathcal{N}_{sol} > 1$, one chooses $q^{\ell_1+\ell_2+\ell_3} > \mathcal{N}_1$ to reduce the time complexity, so $q^{\ell_1+\ell_2+\ell_3} \geq \mathcal{N}_1$) it follows that $p_2\mathcal{N}'_{sol} \leq 2^{-0.0469m}$ is very small. So the assumption is again justified in these cases.

Finally consider the third level of the recursion. The argument is the same as above. Let $\mathcal{N}''_{sol} = \mathcal{N}_{sol}\left(\frac{\mathcal{N}_1}{q^{\ell_1+\ell_2+\ell_3}}\right)\left(\frac{\mathcal{N}_2}{q^{\ell_2+\ell_3}}\right)^2$ be the number solutions to be split. Let $p_3 = \frac{\mathcal{N}_3}{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}}$. For all values α, β, γ listed in Tables 2 and 3 we have $\max(p_3) = 2^{-0.2123m}$. Since $\mathcal{N}''_{sol} \leq \mathcal{N}_{sol} \leq 2^{0.2m}$ (when $\mathcal{N}_{sol} = 1$, one chooses $q^{\ell_1+\ell_2+\ell_3} \approx \mathcal{N}_1$ and $q^{\ell_2+\ell_3} \approx \mathcal{N}_2$, while if $\mathcal{N}_{sol} > 1$, one chooses $q^{\ell_1+\ell_2+\ell_3} \geq \mathcal{N}_1$ and $q^{\ell_2+\ell_3} \geq \mathcal{N}_2$ to reduce the time complexity), $p_3\mathcal{N}''_{sol} = 2^{-0.0123m}$ is very small. Thus $\frac{1-(1-p_3)^{\mathcal{N}''_{sol}}}{p_3}\mathcal{N}_3 \approx \mathcal{N}''_{sol}\mathcal{N}_3$, i.e., almost all the splittings of all the $\mathcal{N}''_{sol}$ solutions at this stage are distinct.

Hence, when the density is $\leq 2^{0.2m}$, Figure 1 is an accurate view of the complexity of the HGJ and BCJ algorithms. When the density is greater than $2^{0.2m}$ then the results of Figure 1 are less rigorous, but they at least give some intuition for how the HGJ and BCJ algorithms behave. In any case, once the density reaches around $2^{0.3m}$ one would likely switch to the CPW algorithm.

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 $\mathbf{x}^T = (\mathbf{x}_0^T|\mathbf{x}_1^T)$ where $\mathbf{x}_0$ has length n and $\mathbf{x}_1$ has length $m-n$ we have that

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

Hence, the Hermite normal form converts an ISIS instance to an instance of LWE (learning with errors) having an extremely small number of samples ($n < m$) and with errors chosen from $\mathcal{B}$ rather than from a discrete Gaussian distribution on $\mathbb{Z}$. We rename $(\mathbf{A}', \mathbf{s}', \mathbf{x}_0, \mathbf{x}_1)$ as $(\mathbf{A}, \mathbf{s}, \mathbf{e}, \mathbf{x})$ so the problem becomes $\mathbf{s} = \mathbf{Ax} + \mathbf{e}$.

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 [12] to be an interesting problem (they called it the "approximate knapsack problem"). Our approach is to replace exact equality of elements in quotient groups $G/H = \mathbb{Z}_q^\ell$, in certain parts of the algorithm, 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 $\mathbf{0}$. 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, letters H_i 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 $\mathcal{E}_1, \mathcal{E}_2$. Recall that we are trying to solve $\mathbf{s} \equiv \mathbf{Ax} + \mathbf{e}$ 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 typically $\mathcal{E}^{(i)} = \mathcal{B}^{\ell_i}$ (For example $\mathcal{E}^{(i)} = \{0,1\}^{\ell_i}$ for HGJ or $\{-1,0,1\}^{\ell_i}$ for BCJ).

Let $H^\flat, H, H^\sharp$ be subgroups of G that denote subgroups used in the CPW/HGJ/BCJ algorithms. We are merging, modulo H, partial solutions modulo $H^\flat \cap H$. For clarity, let us write $G/H^\flat = \mathbb{Z}_q^{\ell'}$ and $G/(H \cap H^\flat) = \mathbb{Z}_q^{\ell+\ell'}$. The input is a pair of lists L_1 and L_2 that are "partial solutions" modulo $H^\flat$. In other words, they are lists of pair $(\mathbf{x}, \mathbf{Ax})$ such that $\mathbf{Ax} + \mathbf{e} \equiv \mathbf{s} \pmod{H^\flat}$ with $\mathbf{e} \in \mathcal{E}^\flat \subset \mathcal{B}^{\ell'}$ (e.g., $\mathbf{e} \in \mathcal{E}^\flat \subset \{0,1\}^{\ell'}$ for HGJ or $\mathbf{e} \in \mathcal{E}^\flat \subset \{-1,0,1\}^{\ell'}$ for BCJ). The goal is to output a set of solutions to the problem $\mathbf{Ax} + \mathbf{e} \equiv \mathbf{s} \pmod{H \cap H^\flat}$ for $\mathbf{x} \in \mathcal{X}$ and, re-using notation, $\mathbf{e} \in \mathcal{E} \subset \mathcal{B}^{\ell+\ell'}$ (e.g., $\mathbf{e} \in \mathcal{E} \subset \{0,1\}^{\ell+\ell'}$ for HGJ or $\mathbf{e} \in \mathcal{E} \subset \{-1,0,1\}^{\ell+\ell'}$ for BCJ). We write $\mathcal{E}_1^\flat, \mathcal{E}_2^\flat$ for the error sets used in the lists L_1, L_2; $\mathcal{E}$ for the error set for $G/(H^\flat \cap H)$; $\mathcal{E}'$ for the error set corresponding to the elements of G/H. For future steps of the algorithm, the output includes information about $\mathbf{Ax} \pmod{H^\sharp}$. The details are given in Algorithm 2.

Algorithm 2 Approximate merge algorithm

```
INPUT: $L_1 = \{(\mathbf{x}, \mathbf{Ax} \pmod{H}) : \mathbf{Ax} + \mathbf{e} \equiv R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_1, \mathbf{e} \in \mathcal{E}_1^\flat\}$,
       $L_2 = \{(\mathbf{x}, \mathbf{Ax} \pmod{H}) : \mathbf{Ax} + \mathbf{e} \equiv \mathbf{s} - R \pmod{H^\flat}, \mathbf{x} \in \mathcal{X}_2, \mathbf{e} \in \mathcal{E}_2^\flat\}$
OUTPUT: $L = \{(\mathbf{x}, \mathbf{Ax} \pmod{H^\sharp}) : \mathbf{Ax} + \mathbf{e} \equiv \mathbf{s} \pmod{H \cap H^\flat}, \mathbf{x} \in \mathcal{X}, \mathbf{e} \in \mathcal{E}\}$
1: Initialise $L = \{\}$
2: Sort $L_1$ with respect to the second coordinate
3: for $(\mathbf{x}_1, \mathbf{u}) \in L_2$ do
4:     Compute $\mathbf{v} = \mathbf{s} - \mathbf{u} \pmod{H}$
5:     for $(\mathbf{x}_2, \mathbf{u}') \in L_1$ with $\mathbf{v} \approx \mathbf{u}'$ (i.e., $\mathbf{u}' - \mathbf{v} \in \mathcal{E}'$) 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 $\mathbf{u}' - \mathbf{v} = \mathbf{e}$ for some $\mathbf{e} \in \mathcal{E}'$) 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. For each match we check the correctness for the whole quotient $G/(H \cap H^\flat)$.

Lemma 2. *Let $G/H = \mathbb{Z}_q^\ell$ and let $\mathcal{E}' \subseteq \mathbb{Z}_q^\ell$ be an error set for G/H. Algorithm 2 performs*

$$\tilde{O}\left(\#L_1 \log(\#L_1) + \#L_2 \left\lceil \#L_1/(q^\ell/\#\mathcal{E}') \right\rceil\right)$$

operations.

Proof. Sorting L_1 takes $\tilde{O}(\#L_1 \log(\#L_1))$ operations on vectors in $G/H = \mathbb{Z}_q^\ell$.

For each pair $(\mathbf{x}_1, \mathbf{u}) \in L_2$ 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_1 is $\#L_1/q^\ell$. 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 check in line 6 of the algorithm is performed

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

times. The most expensive step takes place in line 7, which only executes when the checks in line 6 pass, but for simplicity we assume the running time is proportional to the above formula. □

In the BCJ application in Section 4.3 we will always have $\#L_1 \geq q^\ell$. Hence we can ignore the ceiling operation in the $\tilde{O}(\#L_1\#L_2/(q^\ell/\#\mathcal{E}'))$ term.

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 $\mathcal{E}, \mathcal{E}_1^\flat, \mathcal{E}_2^\flat, \mathcal{E}'$ can lead to different organisation in the algorithm. For example we may take the possible non-zero positions in $\mathcal{E}_1^\flat$ and $\mathcal{E}_2^\flat$ to be disjoint, then after executing line 5 of Algorithm 2 we always have $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \approx \mathbf{s} \pmod{H^\flat}$, and so in line 6 of Algorithm 2 we only need to check $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \approx \mathbf{s} \pmod{H}$ – 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 $(\mathbf{A}, \mathbf{s})$ and denoting $(\mathbf{x}_1, \mathbf{x}_0)$ as $(\mathbf{x}, \mathbf{e})$): Given $\mathbf{A}, q, \mathbf{s}$ to solve $\mathbf{A}\mathbf{x} + \mathbf{e} = \mathbf{s}$ in $G = \mathbb{Z}_q^n$, where $\mathbf{x}$ has length $m - n$ and $\mathbf{e}$ has length n. We assume the problem has high density, in the sense that there are many pairs $(\mathbf{x}, \mathbf{e}) \in \mathcal{B}^m$ 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 four lists. Lists $L_j^{(0)}$ contain pairs $(\mathbf{x}, \mathbf{A}\mathbf{x})$ (in the case of $L_4^{(0)}$ it is $(\mathbf{x}, \mathbf{A}\mathbf{x} - \mathbf{s})$). Merging $L_1^{(0)}$ and $L_2^{(0)}$ gives a list $L_1^{(1)}$ of pairs $(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_2})$ for $\mathbf{x}_1 \in L_1^{(0)}$ and $\mathbf{x}_2 \in L_2^{(0)}$ such that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \approx 0 \pmod{H_1}$, which means $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \equiv \mathbf{e} \pmod{H_1}$ for some $\mathbf{e} \in \mathcal{B}^{n/3}$. Similarly, $L_2^{(1)}$ is a list of pairs $(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_2})$ for $\mathbf{x}_1 \in L_3^{(0)}$ and $\mathbf{x}_2 \in L_4^{(0)}$ such that $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) - \mathbf{s} \equiv \mathbf{e}' \pmod{H_1}$ 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 $u = (m - n)/k$ and let $\mathcal{X}_j = \{(0, \ldots, 0, x_{(j-1)u+1}, \ldots, x_{ju}, 0, \ldots, 0) \in \mathcal{B}^{m-n}\}$ for $1 \leq j \leq k$. It 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 $1 \leq i \leq t$. 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.

Let $\gamma_i = \ell_i/2^{t-i} = \ell_i/(k/2^i)$. For $1 \leq j \leq k/2^i$ we define error sets $\mathcal{E}_{\gamma_i,j}^{(i)} \subseteq \mathcal{B}^{\ell_i}$ restricted to the quotient group G/H_i and with γ_i error positions as

$$\mathcal{E}_{\gamma_i,j}^{(i)} = \{(0, \ldots, 0, e_{(j-1)\gamma_i+1}, \ldots, e_{j\gamma_i}, 0, 0, \ldots, 0) \in \mathcal{B}^{\ell_i}\}.$$

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

Level 0: Compute lists $L_j^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x} \pmod{H_1}) \in \mathcal{X}_j \times \mathbb{Z}_q^{\ell_1}\}$ for $1 \leq j \leq k - 1$ and $L_k^{(0)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x} - \mathbf{s} \pmod{H_1}) \in \mathcal{X}_k \times \mathbb{Z}_q^{\ell_1}\}$. Note that $\#L_j^{(0)} = \#\mathcal{B}^u = \#\mathcal{B}^{(m-n)/k}$. The cost to compute the initial k lists is $O(\#L_j^{(0)}) = O((\#\mathcal{B})^{(m-n)/k})$ or, to be more precise, the cost is approximately $k \cdot \#L_j^{(0)} \cdot C$ bit operations, 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 $\gamma_1 = \ell_1/(k/2)$. For $j = 1, 2, \cdots, k/2$ the sets $\mathcal{E}^{(1)}_{\gamma_1,j} \in \mathcal{B}^{\ell_1}$ specify the positions that are allowed to contain errors. In other words, for $j = 1, 2, \cdots, k/2 - 1$ we construct the new lists

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

and

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

The probability that two random vectors in $\mathbb{Z}_q^{\ell_1}$ have sum in $\mathcal{E}^{(1)}_{\gamma_1,j}$ is $\#\mathcal{E}^{(1)}_{\gamma_1,j}/q^{\ell_1} = \#\mathcal{B}^{\gamma_1}/q^{\ell_1}$, and so the heuristic expected size of the lists $L^{(1)}_j$ is $\#L^{(0)}_{2j-1}\#L^{(0)}_{2j}\#\mathcal{B}^{\gamma_1}/q^{\ell_1} \approx \#\mathcal{B}^{2(m-n)/k+\gamma_1}/q^{\ell_1}$.

Level $i \geq 2$: 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 $j = 1, 2, \cdots, k/2^i - 1$ we construct the new lists

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

and

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

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

It remains to explain how to perform the merging of the lists using Algorithm 2. 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 [11]). Precisely, let v_i 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 $F : \mathbb{Z}_q \to \mathbb{Z}_{2^{v_i}}$ by $F(x) = \lfloor \frac{x}{q/2^{v_i}} \rfloor$. 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 $\mathbf{A}\mathbf{x}_1 + \mathbf{A}\mathbf{x}_2 + \mathbf{e} = \mathbf{0}$, 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 $-\mathbf{A}\mathbf{x}_1 = (7, 2, 4, 5, 6, 4, 0, 7)^T$ and that we are only considering binary errors on the first 4 coordinates. Then we have $F(-\mathbf{A}\mathbf{x}_1) = (1, 0, 1, 1, 6, 4, 0, 7)$. Suppose now $\mathbf{A}\mathbf{x}_2 = (6, 2, 3, 5, 6, 4, 0, 7)$. Then $F(\mathbf{A}\mathbf{x}_2) = (1, 0, 0, 1, 6, 4, 0, 7)$. By looking at the "borderline" entries of $\mathbf{A}\mathbf{x}_2$ we know that we should also check $(1, 0, 1, 1, 6, 4, 0, 7)$. There is no other value to check, since $F((6, 2, 3, 5) + (1, 1, 0, 1)) = (1, 0, 0, 1)$ and $F((6, 2, 3, 5) + (1, 1, 1, 1)) = (1, 0, 1, 1)$ 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 $\mathrm{Flips}(\mathbf{v}) = \{F(\mathbf{v} + \mathbf{e}) : \mathbf{e} \in \mathcal{E}^{(i)}_{\gamma_i,j}\}$, where i, 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 $\mathbf{v}$. It is important to realise that one does not need to loop over all $\mathbf{e} \in \mathcal{E}^{(i)}_{\gamma_i,j}$ to compute $\mathrm{Flips}(\mathbf{v})$; instead one only needs to check entries of $\mathbf{v}$ that are on the borderline. As we will see, the set $\mathrm{Flips}(\mathbf{v})$ is usually quite small. Let p_{flip} be the probability that a randomly chosen element of $\mathbb{Z}_q$ has hash value that flips when adding an error.

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

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 $F(-\mathbf{A}\mathbf{x}_1)$, and these should be stored in a list associated with that entry of the hash table or binary tree representing the sorted list. 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 each $\mathbf{u} \in \text{Flips}(\mathbf{v})$, for occurrences of $\mathbf{u}$ 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_i most significant bits, is $2^{v_i\gamma_i} q^{\ell_i - \gamma_i}$. Hence, the average number of values in the list $L_{2j-1}^{(i-1)}$ that take a given hash value is $\#L_{2j-1}^{(i-1)}/(2^{v_i\gamma_i} q^{\ell_i-\gamma_i})$. Finally, for all good matches we need to compute $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_i}$.

Success Probability. Now we give a heuristic analysis of the algorithm. The algorithm succeeds if $L_1^{(t)}$ is not empty, where $k = 2^t$ is the number of lists used. Denote by $L^{(i)}$ any of the lists at the i-th stage of the algorithm. The heuristic expected size of the lists on each level are

$$\#L^{(0)} = (\#\mathcal{B})^{(m-n)/2^t},$$

$$\#L^{(i)} = \#L^{(i-1)} \#L^{(i-1)} \#\mathcal{B}^{\gamma_i}/q^{\ell_i}, \text{ for all } 1 \le i \le t.$$

Hence, to have $\#L^{(t)} \approx 1$ we want the lists $L^{(1)}, L^{(2)}, \cdots, L^{(t-1)}$ to all be roughly the same size. We make the following deductions.

$$\#L^{(i)} \approx (\#\mathcal{B})^{(m-n)/2^t} \;\Rightarrow\; 1 \approx (\#\mathcal{B})^{(m-n)/2^t + \gamma_i}/q^{\ell_i}, \text{ for all } 1 \le i \le t-1,$$

$$\#L_1^{(t)} \approx 1 \;\Rightarrow\; 1 \approx (\#\mathcal{B})^{2(m-n)/2^t + \gamma_t}/q^{\ell_t}.$$

Following Minder and Sinclair we use an integer program to get optimal values for ℓ_i. We write $\#L^{(i)} = 2^{b_i}$ where $b_0 = (m-n)\log_2(\#\mathcal{B})/2^t$ and hence get $b_i = 2b_{i-1} - \ell_i \log_2(q) + \gamma_i \log_2(\#\mathcal{B})$ where $\gamma_i = \ell_i/2^{t-i}$. The time complexity is proportional to $\max_{0 \le i \le t} \#L^{(i)}$. So to minimize the time one should choose $k = 2^t$ to be as large as possible and then choose the ℓ_i to be a solution to the following integer program.

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

Complexity Analysis. We now fix a choice of $\ell_1, \ell_2, \cdots, \ell_t$ subject to the constraints mentioned above. Our aim is to give a precise formula for the running time of the CPW algorithm using our approximate merge algorithm. This is important as the "approximate 2-merge" is more complicated and takes more time than the "basic 2-merge", and so we have to be certain that we have made a positive improvement overall.

Lemma 3. *Let notation be as above. Let C_1 be the number of bit operations to compute F for the γ_i positions that allow errors. Let C_2 be the number of bit operations to compute Flips($\boldsymbol{v}$). Let C_3 be the number of bit* *operations to check that* $\boldsymbol{x}_1 + \boldsymbol{x}_2 \in \mathcal{B}^{m-n}$ *and that* $\boldsymbol{A}(\boldsymbol{x}_1 + \boldsymbol{x}_2) \pmod{H_i} \in \mathcal{E}^{(i)}_{\gamma_i,j}$ *(we only need to check* γ_i *positions of the error). Then the i-th iteration of Algorithm 2 requires*

$$\#L^{(i-1)}_{2j-1}\left(C_1 + \log_2(\#L^{(i-1)}_{2j-1})\log_2(q^{\ell_i})\right)$$
$$+\ \#L^{(i-1)}_{2j}\left(C_2 + 2^{\gamma_i p_{\mathit{flip}}}\left[\log_2(\#L^{(i-1)}_{2j-1})\log_2(q^{\ell_i})\ +\ (\#L^{(i-1)}_{2j-1}/(2^{v_i\gamma_i}q^{\ell_i-\gamma_i}))C_3\right]\right)$$
$$+\ \#L^{(i)}_{j}(\ell_{i+1}(m-n)/2^{t-i}\log_2(q))$$

bit operations.

Proof. The first operation involves sorting $L^{(i-1)}_{2j-1}$, where we use F to compute the value $\mathbf{u}$ that acts as the index to the hash table or binary tree. The total cost is $\#L^{(i-1)}_{2j-1}(C_1 + \log_2(\#L^{(i-1)}_{2j-1})\log_2(q^{\ell_i})$ bit operations.

Next, for each pair $(\mathbf{x}_2, \mathbf{v}) \in L^{(i-1)}_{2j}$ one needs to compute Flips($\mathbf{v}$). For each element in Flips($\mathbf{v}$) (the expected number is $2^{\gamma_i p_{\text{flip}}}$) one needs to look up the value in the sorted list. For each match one has, on average, $\#L^{(i-1)}_{2j-1}/(2^{v_i\gamma_i}q^{\ell_i-\gamma_i})$ different values from the first list with that hash value. For each of these we need to compute $\mathbf{x}_1 + \mathbf{x}_2$ and $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_i}$.

Finally, one needs to compute the values $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2) \pmod{H_{i+1}}$ for the next iteration. This occurs only for the "good" values, of which there are by definition $\#L^{(i)}_j$. The computation of $\mathbf{A}(\mathbf{x}_1 + \mathbf{x}_2)$ is adding at most $(m-n)/2^{t-i}$ vectors of length ℓ_{i+1}. □

Larger values for v_i increase p_{flip} but reduce $\#L^{(i-1)}_{2j-1}/(2^{v_i\gamma_i}q^{\ell_i-\gamma_i})$. So we choose v_i to balance the costs $2^{\gamma_i p_{\text{flip}}}$ and $\#L^{(i-1)}_{2j-1}/(2^{v_i\gamma_i}q^{\ell_i-\gamma_i})$. Hence, the time to perform the "approximate 2-merge" can be proportional to $\#L^{(i-1)}_{2j-1}\log_2(\#L^{(i-1)}_{2j-1})$. When $\gamma_i = 0$ we recover the basic merge algorithm and so $C_1 = 0$, $C_2 = 0$ (and, when using the algorithm in the CPW context, $C_3 = 0$), $v_i = \log_2(q)$, $2^{\gamma_i p_{\text{flip}}} = 1$, and $\#L^{(i-1)}_{2j-1}/(2^{v_i\gamma_i}q^{\ell_i-\gamma_i}) = \#L^{(i-1)}_{2j-1}/q^{\ell_i}$.

As in the work of Minder and Sinclair, it is not possible to give a general complexity statement in terms of $(m, n, q, \mathcal{B})$, since the complexity depends on the choice of parameters in the algorithm, and they are the solution to the integer program. However, in Section 5 we use the algorithm to attack the SWIFFT hash function and demonstrate improved running times compared with the standard CPW algorithm.

4.3 The HGJ and BCJ algorithm

Recall that our problem is: Given $\mathbf{A}, \mathbf{s}$ to solve $\mathbf{Ax} + \mathbf{e} = \mathbf{s}$ in $G = \mathbb{Z}_q^n$, where $\mathbf{x}$ has length $m' = m - n$ and $\mathbf{e}$ has length n. Here $\mathbf{A}$ is an $n \times m'$ matrix, $\mathbf{x} \in \{0,1\}^{m'}$ and $\mathbf{e} \in \{0,1\}^n$. We assume wt($\mathbf{x}$) $= m'/2$ and wt($\mathbf{e}$) $= n/2$. Assume for simplicity that the density is approximately 1 (hence $2^{m'+n}/q^n \approx 1$)[8]. Using Algorithm 2 we can get an improved version of the HGJ/BCJ algorithm. Since HGJ is a special case of BCJ, we discuss the general case.

We use the BCJ algorithm with 3 levels of recursion to find a solution $\mathbf{x} \in \{0,1\}^{m'}$ of weight $m'/2$. 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. With this notation, our desired solution is an element of $\mathcal{X}_{1/2,0}$.

Following [2] we choose suitable parameters α, β, γ (see below). The first level of recursion splits $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_{1/4+\alpha,\alpha}$. The second level of recursion splits $\mathbf{x} \in \mathcal{X}_{1/4+\alpha,\alpha}$ into $\mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$. The third level of recursion splits a vector into a sum of two vectors in the set $\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}$.

We also use the BCJ idea to split the errors. This is a little more complex to describe as the error sets are in $\{-1,0,1\}^{n'}$ for varying values of $n' < n$. The notation $\mathcal{E}_{a,b}$ will mean a set of vectors in $\{-1,0,1\}^{n'}$

[8] We do not analyse this algorithm in the case of density > 1, but it is clear that similar improvements can be obtained as in Section 3. As in Section 3 it is not possible to give a general formula for the running time.

with an' entries equal to 1 and bn' entries equal to -1. We assume $\mathbf{e} \in \mathcal{E}_{1/2,0}$ initially ($n' = n$). We will fix parameters α', β', γ' (to be specified below).

Let H_1, H_2, H_3 be subgroups of G, of the usual form, such that $\#(G/H_i) = q^{\ell_i}$. For $1 \le i \le 3$ we define $\mathcal{E}_{a,b}^{(i \to 3)} \subseteq \{-1,0,1\}^{n'}$ where $n' = \ell_i + \cdots + \ell_3$ to be the set of vectors with an' entries equal to 1 and bn' entries equal to -1. These are considered as subsets of the quotient groups $G/(H_i \cap \cdots \cap H_3)$. We denote $\mathcal{E}_{a,b}^{(3\to 3)}$ by $\mathcal{E}_{a,b}^{(3)}$.

Algorithm. The first level of recursion chooses a random value $R_1 \in \mathbb{Z}_q^{\ell_1+\ell_2+\ell_3} = G/(H_1 \cap H_2 \cap H_3)$ and computes two lists $L_1^{(1)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_{1/4+\alpha,\alpha}, \mathbf{A}\mathbf{x} + \mathbf{e} \equiv R_1 \pmod{H_1 \cap H_2 \cap H_3}$ for some $\mathbf{e} \in \mathcal{E}^{(1\to 3)}_{1/4+\alpha',\alpha'}\}$ and $L_2^{(1)}$ which is the same except $\mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{s} - R_1 \pmod{H_1 \cap H_2 \cap H_3}$.

The second level of recursion computes $L_1^{(1)}$ and $L_2^{(1)}$ by splitting into further lists. We split $\mathbf{x} \in \mathcal{X}_{1/4+\alpha,\alpha}$ into $\mathbf{x}_1 + \mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$ and split error vectors into $\mathcal{E}^{(2\to 3)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}$. For example, choosing a random value R_2, $L_1^{(1)}$ is split into $L_1^{(2)} = \{(\mathbf{x}, \mathbf{A}\mathbf{x}) : \mathbf{x} \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}, \mathbf{A}\mathbf{x} + \mathbf{e} \equiv R_2 \pmod{H_2 \cap H_3}$, for some $\mathbf{e} \in \mathcal{E}^{(2\to 3)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}\}$ and $L_2^{(2)}$ is similar except the congruence is $\mathbf{A}\mathbf{x} + \mathbf{e} \equiv R_1 - R_2 \pmod{H_2 \cap H_3}$.

The final level of recursion computes each list $L_j^{(2)}$ by splitting $\mathbf{x} \in \mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$ into a sum $\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}$ and splitting the error $\mathbf{e} \pmod{H_3} \in \mathcal{E}^{(3)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}$ into a sum from $\mathcal{E}^{(3)}_{1/16+\alpha'/4+\beta'/2+\gamma',\alpha'/4+\beta'/2+\gamma'}$. The lists at the final level are computed using the Shroeppel-Shamir algorithm.

Success Probability. Suppose there is a unique solution $(\mathbf{x}, \mathbf{e}) \in \{0,1\}^{m'+n}$ to the ISIS instance, where $\mathbf{x}$ has weight $m'/2$ and $\mathbf{e}$ has weight $n/2$. 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 together with a splitting $\mathbf{e} \equiv \mathbf{e}_1 + \mathbf{e}_2 \pmod{H_1 \cap H_2 \cap H_3}$, so that for a randomly chosen $R_1 \in G/(H_1 \cap H_2 \cap H_3)$, $\mathbf{A}\mathbf{x}_1 + \mathbf{e}_1 \equiv R_1 \pmod{H_1 \cap H_2 \cap H_3}$ and $\mathbf{A}\mathbf{x}_2 + \mathbf{e}_2 \equiv \mathbf{s} - R_1 \pmod{H_1 \cap H_2 \cap H_3}$. The number of ways to split $\mathbf{x}$ (with length m') on the first level of recursion is

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

where the notation $\binom{N}{a,b,(N-a-b)} = \binom{N}{a,b} = \binom{N}{a}\binom{N-a}{b}$ denotes the usual multinomial coefficient. The number of ways to split $\mathbf{e} \pmod{H_1 \cap H_2 \cap H_3}$ (with length $\ell_1 + \ell_2 + \ell_3$) on the first level of recursion is

$$\mathcal{N}_1' = \binom{(\ell_1+\ell_2+\ell_3)/2}{(\ell_1+\ell_2+\ell_3)/4}\binom{(\ell_1+\ell_2+\ell_3)/2}{\alpha'(\ell_1+\ell_2+\ell_3), \alpha'(\ell_1+\ell_2+\ell_3)}.$$

For randomly chosen $R_1 \in G/(H_1 \cap H_2 \cap H_3)$, we expect there to be a valid splitting if $\mathcal{N}_1\mathcal{N}_1' \ge q^{\ell_1+\ell_2+\ell_3}$. Indeed, the expected number of valid splittings should be roughly $\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3}$. Hence, we choose $\mathcal{N}_1\mathcal{N}_1' \approx q^{\ell_1+\ell_2+\ell_3}$ to make sure a valid splitting pair exists at this stage with significant probability.

For the second stage we assume that we already made a good choice in the first stage, and indeed that we have $\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3}$ possible values for $(\mathbf{x}_1, \mathbf{e}_1)$ where $\mathbf{x}_1 \in \mathcal{X}_{1/4+\alpha,\alpha}, \mathbf{e}_1 \in \mathcal{E}^{(1\to 3)}_{1/4+\alpha',\alpha'}$. The number of ways to further split $\mathbf{x}_1$ is

$$\mathcal{N}_2 = \binom{(1/4+\alpha)m'}{(1/8+\alpha/2)m'}\binom{\alpha m'}{\alpha m'/2}\binom{(3/4-2\alpha)m'}{\beta m', \beta m', (3/4-2\alpha-2\beta)m'}.$$

The number of ways to further split $\mathbf{e}_1 \pmod{H_2 \cap H_3}$ is $\mathcal{N}_2'$ is

$$\binom{(1/4+\alpha')(\ell_2+\ell_3)}{(1/8+\alpha'/2)(\ell_2+\ell_3)}\binom{\alpha'(\ell_2+\ell_3)}{\alpha'(\ell_2+\ell_3)/2}\binom{(3/4-2\alpha')(\ell_2+\ell_3)}{\beta'(\ell_2+\ell_3), \beta'(\ell_2+\ell_3), (3/4-2\alpha'-2\beta')(\ell_2+\ell_3)}.$$

The expected number of valid splittings at this stage should be roughly

$$(\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2\mathcal{N}_2'/q^{\ell_2+\ell_3})^2.$$

For a randomly chosen $R_2 \in G/(H_2 \cap H_3)$, there is a good chance that a valid splitting exists if

$$(\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2\mathcal{N}_2'/q^{\ell_2+\ell_3})^2 \geq 1$$

(recall that at this stage we need to split both $\mathbf{x}_1$ and $\mathbf{x}_2$). Hence, since we already choose $\mathcal{N}_1\mathcal{N}_1' \approx q^{\ell_1+\ell_2+\ell_3}$, we now impose the condition $\mathcal{N}_2\mathcal{N}_2' \approx q^{\ell_2+\ell_3}$.

In the final stage (again assuming good splittings in the entire second stage), the number of ways to split elements in $\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}$ is

$$\mathcal{N}_3 = \binom{(1/8+\alpha/2+\beta)m'}{(1/16+\alpha/4+\beta/2)m'}\binom{(\beta+\alpha/2)m'}{(\beta/2+\alpha/4)m'}\binom{(7/8-\alpha-2\beta)m'}{\gamma m', \gamma m', (7/8-\alpha-2\beta-2\gamma)m'}.$$

The number of ways to split elements in $\mathcal{E}^{(3)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}$ is:

$$\mathcal{N}_3' = \binom{(1/8+\alpha'/2+\beta')\ell_3}{(1/16+\alpha'/4+\beta'/2)\ell_3}\binom{(\beta'+\alpha'/2)\ell_3}{(\beta'/2+\alpha'/4)\ell_3}\binom{(7/8-\alpha'-2\beta')\ell_3}{\gamma'\ell_3, \gamma'\ell_3, (7/8-\alpha'-2\beta'-2\gamma')\ell_3}.$$

The expected number of valid splittings is

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

which we require to be ≥ 1. Hence, we add the additional constraint $\mathcal{N}_3\mathcal{N}_3' \approx q^{\ell_3}$. Thus, choosing $\#G/H_3$ close to $\mathcal{N}_3\mathcal{N}_3'$, $\#G/(H_2 \cap H_3)$ close to $\mathcal{N}_2\mathcal{N}_2'$ and $\#G/(H_1 \cap H_2 \cap H_3)$ close to $\mathcal{N}_1\mathcal{N}_1'$ then the heuristic success probability of the algorithm should be significantly noticeable.

Parameters. Suitable parameters need to be chosen to optimize the running time and have good success probability.

As in previous sections, we use the estimate $\#\mathcal{X}_{a,b} \approx 2^{H(a,b)\cdot m'}$ where

$$H(x,y) = -x\log_2(x) - y\log_2(y) - (1-x-y)\log_2(1-x-y)$$

and similarly for $\mathcal{E}_{a,b}$. We also use

$$h(x) = -x\log_2(x) - (1-x)\log_2(1-x)$$

to estimate the binomial. Then the logarithm of the number of decompositions for each level is

$$\begin{aligned}
\log_2(\mathcal{N}_1) &\approx \tfrac{m'}{2}\left(h(1/2) + H(2\alpha, 2\alpha)\right) \approx m'\frac{1+H(2\alpha,2\alpha)}{2},\\
\log_2(\mathcal{N}_2) &\approx m' \cdot \frac{1+8\alpha+(3-8\alpha)\cdot H(\frac{\beta}{3/4-2\alpha}, \frac{\beta}{3/4-2\alpha})}{4},\\
\log_2(\mathcal{N}_3) &\approx m' \cdot \frac{1+8\alpha+16\beta+(7-8\alpha-16\beta)\cdot H(\frac{\gamma}{7/8-\alpha-2\beta}, \frac{\gamma}{7/8-\alpha-2\beta})}{8}.
\end{aligned}$$

The (logarithmic) number of decompositions for the error vector is similar.

$$\begin{aligned}
\log_2(\mathcal{N}_1') &\approx (\ell_1+\ell_2+\ell_3)\cdot\frac{1+H(2\alpha',2\alpha')}{2}\\
\log_2(\mathcal{N}_2') &\approx (\ell_2+\ell_3)\cdot\frac{1+8\alpha'+(3-8\alpha')\cdot H(\frac{\beta'}{3/4-2\alpha'}, \frac{\beta'}{3/4-2\alpha'})}{4}\\
\log_2(\mathcal{N}_3') &\approx \ell_3\cdot\frac{1+8\alpha'+16\beta'+(7-8\alpha'-16\beta')\cdot H(\frac{\gamma'}{7/8-\alpha'-2\beta'}, \frac{\gamma'}{7/8-\alpha'-2\beta'})}{8}.
\end{aligned}$$

Assuming that $2^{m'+n} \approx q^n$, we have $n \approx \frac{m'}{\log_2 q-1}$, and we choose the following parameters (these are the result of an optimisation problem to minimise the running time of the algorithm).

$$\begin{aligned}
&\alpha = 0.0267, \beta = 0.0168, \gamma = 0.0029;\\
&\alpha' = 0.0279, \beta' = 0.0027, \gamma' = 0.0027;\\
&q^{\ell_1} = 2^{0.2673m'}2^{0.4558n}, q^{\ell_2} = 2^{0.2904m'}2^{0.1388n} \text{ and } q^{\ell_3} = 2^{0.2408m'}2^{0.0507n}.
\end{aligned}$$

For such parameters, we can estimate

$$\mathcal{N}_1 \approx 2^{0.7985m'},\ \mathcal{N}_2 = 2^{0.5312m'}, \text{ and } \mathcal{N}_3 = 2^{0.2408m'};$$

and

$$\mathcal{N}_1' = 2^{0.8082(\ell_1+\ell_2+\ell_3)},\ \mathcal{N}_2' = 2^{0.3568(\ell_2+\ell_3)} \text{ and } \mathcal{N}_3' = 2^{0.2106\ell_3}.$$

We consider the success probability. The number of splittings for the first level is

$$\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3}.$$

With the above parameters, then the logarithm (base 2) of this value is approximately

$$\begin{aligned}
&0.7985\,m' - \frac{0.7985\,m'^2}{m'+n} - \frac{0.7985\,m'n}{m'+n} - \frac{0.1238\,n^2}{m'+n}\\
\approx\ & -\frac{0.1238\,n^2}{m'+n}
\end{aligned}$$

Similarly, the logarithm of the number of valid splittings for the second stage is

$$\log_2\left((\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2\mathcal{N}_2'/q^{\ell_2+\ell_3})^2\right) \approx -\frac{0.3676\,n^2}{m'+n}.$$

Finally, the logarithm of the number of valid splittings for the last stage is

$$\log_2\left((\mathcal{N}_1\mathcal{N}_1'/q^{\ell_1+\ell_2+\ell_3})(\mathcal{N}_2\mathcal{N}_2'/q^{\ell_2+\ell_3})^2(\mathcal{N}_3\mathcal{N}_3'/q^{\ell_3})^4\right) \approx -\frac{0.5278\,n^2}{m'+n}.$$

For example, taking $m = 1024$, $n = 64$ and $m' = m - n$ gives expected number of splittings at each level to be $0.7, 0.36$ and 0.23 respectively, giving a total success probability of approximately 0.06. So we expect to need to repeat the algorithm around 16 times. For fixed n and letting $m' \to \infty$ then the success probability tends to 1.

Running time. We now consider the running time for one execution of the algorithm. (We do not consider the number of repetitions requried due to the success probability here.)

Theorem 2. *Let $\boldsymbol{s} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{e}$ in $\mathbb{Z}_q^n$ where $\boldsymbol{x} \in \{0,1\}^{m'}$ has weight $m'/2$, and $\boldsymbol{e} \in \{0,1\}^n$ has weight $n/2$. Suppose the density is approximately 1, so that $2^m = 2^{m'+n} \approx q^n$. With notation as above, and assuming heuristics about the pseudo-randomnes of $\boldsymbol{A}\boldsymbol{x}+\boldsymbol{e}$, the approximate BCJ algorithm runs in $\tilde{O}(2^{0.2912m'+0.1899n})$ time.*

Proof. We run the algorithm as described using the parameters specified above.
First level. In the first level of the recursion, the expected size of each list is given by

$$\begin{aligned}
\log_2\left(\#L^{(1)}\right) &= \log_2\left(\#\mathcal{X}_{1/4+\alpha,\alpha}\,\#\mathcal{E}^{(1\to3)}_{1/4+\alpha',\alpha'}\,/\,q^{\ell_1+\ell_2+\ell_3}\right)\\
&\approx m'\,H(1/4+\alpha,\alpha) + H(1/4+\alpha',\alpha')(\ell_1+\ell_2+\ell_3) - (\ell_1+\ell_2+\ell_3)\log_2(q).
\end{aligned}$$

For our parameters we have the relation

$$q^{\ell_1+\ell_2+\ell_3} = 2^{0.7985m'}2^{0.6453n}.$$

Hence the logarithmic size is

$$\begin{aligned}\log_2\left(\#L^{(1)}\right) =& 0.2173m' + 0.8169\frac{m'n}{m'+n} - 0.6453n + 0.6601\frac{n^2}{m'+n}\\ &\leq 0.2173m' + 0.1716n.\end{aligned}$$

Using the approximate merge (Algorithm 2) this requires time

$$T_1 = \tilde{O}\left(\#L^{(1)}\#L^{(1)}2^{n-\ell_1-\ell_2-\ell_3}/q^{n-\ell_1-\ell_2-\ell_3}\right) \tag{6}$$

One can compute a precise formula for $\log_2(T_1)$ by evaluating equation (6) with our parameters. We use a computer script to keep track of all the quantities. One can show that $\log_2(T_1) \leq 0.2331m' + 0.1899n$.
Second level. Similarly, in the second level of recursion the size of the lists is approximately

$$\begin{aligned}\log_2\left(\#L^{(2)}\right) =& \log_2\left(\#\mathcal{X}_{1/8+\alpha/2+\beta,\alpha/2+\beta}\,\#\mathcal{E}^{(2\to3)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}\,/\,q^{\ell_2+\ell_3}\right)\\ &\approx m'\,H(1/8+\alpha/2+\beta,\alpha/2+\beta)\\ &\quad + H(1/8+\alpha'/2+\beta',\alpha'/2+\beta')(\ell_2+\ell_3) - (\ell_2+\ell_3)\log_2(q).\end{aligned}$$

For our parameters we have the relation

$$q^{\ell_2+\ell_3} = 2^{0.5312m'}2^{0.1895n}.$$

Hence the logarithmic size is

$$\begin{aligned}\log_2\left(\#L^{(2)}\right) =& 0.2791\,m' + \frac{0.3756\,m'n}{m'+n} - 0.1895\,n + \frac{0.1340\,n^2}{m'+n}\\ &\leq 0.2791m' + 0.1861n.\end{aligned}$$

Using the approximate merge (Algorithm 2), the cost to merge is

$$T_2 = \tilde{O}\left(\#L^{(2)}\#L^{(2)}\#\mathcal{E}^{(1)}_{1/4+\alpha',\alpha'}/q^{\ell_1}\right).$$

With above parameters

$$\begin{aligned}\log_2(T_2) \approx& 0.2908\,m' + \frac{1.0246\,m'n}{m'+n} - 0.8348\,n + \frac{0.7342\,n^2}{m'+n}\\ &\leq 0.2908m' + 0.1898n.\end{aligned}$$

Third level. The last level of recursion computes lists of expected size

$$\begin{aligned}\log_2\left(\#L^{(3)}\right) &= \log_2\left(\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}\right)\\ &\quad + \log_2\left(\#\mathcal{E}^{(3)}_{1/16+\alpha'/4+\beta'/2+\gamma',\alpha'/4+\beta'/2+\gamma'}\right) - \ell_3\log_2(q)\\ &\approx m'\,H(1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma)\\ &\quad + \ell_3\,H(1/16+\alpha'/4+\beta'/2+\gamma',\alpha'/4+\beta'/2+\gamma') - \ell_3\log_2(q).\end{aligned}$$

For our parameters we have the relation

$$q^{\ell_3} = 2^{0.2408m'}2^{0.0507n}.$$

Hence the logarithmic size is

$$\begin{aligned}\log_2\left(\#L^{(3)}\right) =& 0.2908\, m' + \frac{0.1120\, m'n}{m'+n} - 0.0507\, n + \frac{0.0236\, n^2}{m'+n} \\ \leq& \ 0.2908 m' + 0.0613 n.\end{aligned}$$

Using the approximate merge (Algorithm 2), the cost to merge is

$$T_3 = \tilde{O}\left(\#L^{(3)}\#L^{(3)}\mathcal{E}^{(2)}_{1/8+\alpha'/2+\beta',\alpha'/2+\beta'}/q^{\ell_2}\right).$$

With above parameters

$$\begin{aligned}\log_2(T_3) \approx& 0.2912\, m' + \frac{0.4293\, m'n}{m'+n} - 0.2402\, n + \frac{0.1453\, n^2}{m'+n} \\ \leq& \ 0.2912\, m' + 0.1891 n.\end{aligned}$$

Each list in the last level can be constructed using the Shroeppel-Shamir algorithm (or its variants) in time

$$T_4 = \tilde{O}(\sqrt{\#\mathcal{X}_{1/16+\alpha/4+\beta/2+\gamma,\alpha/4+\beta/2+\gamma}\ \#\mathcal{E}^{(3)}_{1/16+\alpha'/4+\beta'/2+\gamma',\alpha'/4+\beta'/2+\gamma'}}).$$

We have

$$\begin{aligned}\log_3(T_4) \approx& 0.2658\, m' + \frac{0.0560\, m'n}{m'+n} + \frac{0.0118\, n^2}{m'+n} \\ \leq& \ 0.2658\, m' + 0.0560 n.\end{aligned}$$

By taking the maximum of the running times of all levels in the recursion, the time complexity of the algorithm is $\tilde{O}(2^{0.2912m'+0.1899n})$. □

Remark 1. Note that $\tilde{O}(2^{0.2912m'+0.1899n}) < \tilde{O}(2^{0.2912m})$, so the algorithm is an improvement over standard BCJ by a multiplicative factor $2^{0.1013n}$.

Corollary 1. *Let n be fixed and $m \to \infty$. One can solve $(m, n, q, \{0,1\})$-ISIS in $\tilde{O}(2^{0.2912m-0.1013n})$ bit operations.*

Proof. Since n is fixed the success probability is constant. The running time is therefore $\tilde{O}(2^{0.2912(m-n)+0.1899n})$. □

For fixed n, Corollary 1 is the same $\tilde{O}(2^{0.2912m})$ statement as given in [2], but with an improved constant.

5 Improved attacks on SWIFFT

We present improved attacks for both inversion and finding collisions of the SWIFFT hash function. The parameters are $m = 1024, n = 64, q = 257$ and $\mathcal{B} = \{0, 1\}$. Note that we ignore the ring structure and just treat SWIFFT as an instance of ISIS. The best previously known attacks for the SWIFFT inversion and collision problems require 2^{148} bit operations and 2^{113} bit operations respectively. Our improved attacks solve the SWIFFT inversion and collision problems in 2^{138} bit operations and 2^{104} bit operations respectively.

5.1 Inverting SWIFFT.

Lyubashevsky, Micciancio, Peikert and Rosen [17] discussed using the original CPW algorithm to solve the $(1024, 64, 257, \{0, 1\})$-SIS problem: "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 compute 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 Lemma 3, we predict the total running time to be approximately 2^{148} bit operations.

We now show that using the Hermite normal form 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 we compute the Hermite normal form, to reduce $\mathbf{A}$ to a 64×936 matrix. We then use $k = 8$ to break $\{0,1\}^{936}$ into 8 groups of length 117. Let $\ell_1 = 15, \ell_2 = 16$ and $\ell_3 = 33$. Construct 8 initial lists of size 2^{117}.

At each step, we merge two lists in a similar way to the original CPW algorithm but using Algorithm 2. At the bottom we merge the initial eight 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[9] 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 $v_1 = 7$ 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 $\frac{2^{117}}{2^{v_1 \gamma_1} q^{\ell_1 - \gamma_1}} \leq 2$.

At the next level we merge the four lists 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 two 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 $v_2 = 7$.

At the top level we merge the two lists of sizes $2^{114.75}$ 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[10] is $\frac{2^{114.75+115.75} 2^{33}}{257^{33}} \approx 2^{-0.7}$, we use $v_3 = 4$ for the hashing.

In conclusion, the maximum size of the lists at each level is $2^{117.92}$, and using Lemma 3 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 [17] 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 [17] that the lists have 2^{102} elements at each level is implausible. This is permitted in their context, since their goal is simply to get a lower bound on the running time. But we find it useful to obtain a more accurate estimate of the running time of this approach. 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 increase the success probability by using Minder and Sinclair's refinement of CPW [20]. For $k = 16$ lists one can take $\ell_1 = 12, \ell_2 = 14, \ell_3 = 12, \ell_4 = 26$. The maximum size of the lists at all the levels is then around 2^{107}. Using Lemma 3 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 [13] (an early version of [12]). The idea is to attack the original $\{0,1\}$-SIS problem directly: first using the original CPW

[9] The analysis of the expected size of lists assumes independence of vectors in the lists. However, when the ISIS instance comes from a ring then there are certain symmetries that show that vectors are not independent. We have theoretical and experimental evidence that these issues do not effect the analysis of the CPW algorithm on SWIFFT. See Section 6 for further discussion.

[10] It's possible to tune certain constraints in 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 have to run the whole algorithm twice.

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 [13] to be proportional to 2^{100}. Lemma 3 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 the HNF 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 $\mathbf{A}\mathbf{x} \pmod{H}$ has coordinates lying in $\{-1, 0\}$ (in other words, there is a binary error vector $\mathbf{e}$ such that $\mathbf{A}\mathbf{x} + \mathbf{e} \equiv \mathbf{0} \pmod{H}$). 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 $b_t \geq 0$ (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 to be large enough to exploit the 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 $2^{83.88}$, so the space is proportional to 2^{84}. By Lemma 3 the total running time is estimated to be 2^{104} bit operations; a 2^9 speed-up over the previous best method.

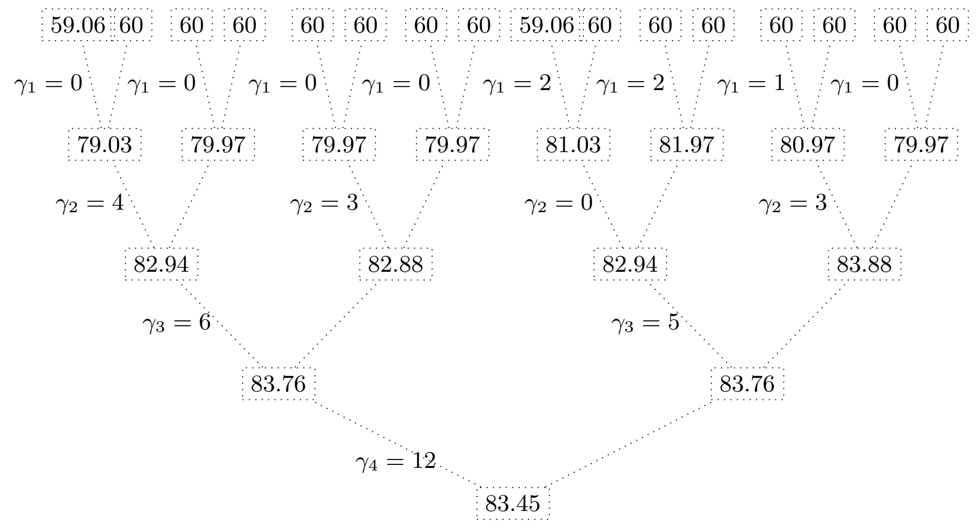

Fig. 2. Parameter choices and list sizes for the approximate-CPW algorithm for finding collisions in SWIFFT. As stated above, we take $\ell_1 = 5, \ell_2 = 10, \ell_3 = 11$ and $\ell_4 = 12$. The values v_i, being the number of most significant bits we use for the hash in the cases when $\gamma_i \neq 0$, are $v_1 = v_2 = 8$ and $v_3 = v_4 = 7$. The numbers in the dotted box denote the $\log_2$ size of the list; the γ_i is used in the approximate-merge algorithm.

6 Ring-SIS

We consider the ring $R_q = \mathbb{Z}_q[t]/(t^n + 1)$, where q is prime and n is typically a power of 2. For example, the SWIFFT hash function is defined using $q = 257$ and $n = 64$. Ring elements are often represented as tuples $(a_0, \dots, a_{n-1}) \in \mathbb{Z}_q^n$ corresponding to the polynomial $a_0 + a_1 t + \cdots + a_{n-1}t^{n-1}$. Define a subset $\mathcal{X}$ of R_q as those polynomials with coefficients $a_i \in \{0, 1\}$, or possibly $\{-1, 0, 1\}$. The *Ring-SIS problem* is, given $\mathbf{a}_1, \dots, \mathbf{a}_m \in R_q$, to find $\mathbf{x}_1 \dots, \mathbf{x}_m \in \mathcal{X}$ such that $\mathbf{0} = \mathbf{a}_1\mathbf{x}_1 + \cdots + \mathbf{a}_m\mathbf{x}_m$ and not all $\mathbf{x}_i = 0$. The SWIFFT hash function uses $m = 16$, giving $2^{mn} = 2^{1024}$ choices for $(\mathbf{x}_1, \dots, \mathbf{x}_m)$. One can find collisions in the SWIFFT hash function by solving Ring-SIS with the set $\mathcal{X}$ being polynomials with coefficients $a_i \in \{-1, 0, 1\}$.

All previous cryptanalytic work on the SWIFFT hash function ignored the ring structure and converted the problem to (I)SIS. In this section we consider exploiting the ring structure to obtain a speedup. Our tool is the map $\psi : R_q \to R_q$ given by $\psi(a(t)) = ta(t)$. This is not a ring homomorphism. We will think of ψ as an invertible linear map on the $\mathbb{Z}_q$-vector space R_q, and call it a "symmetry" (technically, we have a group action). In terms of coefficient vectors, we have

$$\psi(a_0, \dots, a_{n-1}) = (-a_{n-1}, a_0, \dots, a_{n-2}).$$

Note that ψ has order $2n$ and that $\psi^n = -1$.

The key observation is that

$$\psi(\mathbf{a}\mathbf{x}) = \psi(a(t)x(t)) = ta(t)x(t) = a(t)(tx(t)) = \mathbf{a}\psi(\mathbf{x}).$$

Hence, an orbit $\{\psi^i(\mathbf{a}\mathbf{x}) : 0 \le i < 2n\}$ can be considered as $\mathbf{a}$ times the orbit of $\mathbf{x}$. If the set $\mathcal{X}$ has the property that $\psi(\mathcal{X}) = \mathcal{X}$ then one can adapt the CPW algorithm to work with the set of orbits $\mathcal{X}/\psi$ rather than the whole set $\mathcal{X}$. The hope is that this gives a reduction by a factor $2n$ in the cost of all stages of the algorithm.

For the parameters in SWIFFT we do not have $\psi(\mathcal{X}) = \mathcal{X}$. This is because $\mathcal{X}$ consists of polynomials with coefficients in $\{0, 1\}$, whereas $t^{64} = -1$ and so large enough shifts of polynomials in $\mathcal{X}$ have some coefficients equal to -1. So for the remainder of this section we restrict attention to cases where $\psi(\mathcal{X}) = \mathcal{X}$. This is the case when the ring R_q is defined as $\mathbb{Z}_q[t]/(t^n - 1)$, or if $\mathcal{X}$ is the set of polynomials with coefficients in $\{-1, 0, 1\}$.

A remaining non-trivial issue is how to deal with quotients G/H. Our group G is now the additive group of the ring R_q, and to have a well-defined action of ψ on the quotient G/H it is necessary that H be invariant under ψ (i.e., $\psi(H) = H$). It might seem that this condition greatly restricts our possible choices for H, and so does not allow arbitrary fine-tuning of the size of G/H. We now explain that for the parameters used in SWIFFT no such obstruction arises.

The matrix representation of ψ with respect to the basis $\{1, t, \dots, t^{n-1}\}$ is

$$\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -1 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

One can verify that, when $q = 257$ and $n = 64$, this matrix has n distinct eigenvalues (specifically, the 64 distinct 128-th roots of 1 in $\mathbb{Z}_{257}^*$). This happens since $128 \mid (257 - 1)$, a choice that was taken so that $t^{64} + 1$ splits completely modulo q (which allows a nice search-to-decision reduction). This means that there is an eigenbasis for ψ. With respect to this basis of $\mathbb{Z}_{257}^{64}$ the matrix representing ψ is diagonal. Hence, by taking H to be the group generated by any $64 - \ell$ of these eigenvectors we have a ψ-invariant subgroup such that $\#(G/H) = q^\ell$. In other words, the ring chosen for SWIFFT is the "best possible" choice for an attacker: if $t^n + 1$ did not split completely modulo q then the attacker's job would likely be a little more difficult.

We now very briefly describe the CPW algorithm in the case when $\mathcal{X}$ consists of polynomials with coefficients in $\{-1, 0, 1\}$. Suppose one wants to solve $\mathbf{0} = \mathbf{a}_1\mathbf{x}_1 + \cdots + \mathbf{a}_{16}\mathbf{x}_{16}$ using an 8-list algorithm.

The original CPW algorithm would choose a suitable subgroup H_1 and compute lists $L_i^{(0)} = \{(\mathbf{a}_{2i-1}\mathbf{x}_{2i-1} + \mathbf{a}_{2i}\mathbf{x}_{2i} \pmod{H_1}, \mathbf{x}_{2i-1}, \mathbf{x}_{2i}) : \mathbf{x}_{2i-1}, \mathbf{x}_{2i} \in \mathcal{X}\}$ for $1 \le i \le 8$. The size of each set $L_i^{(0)}$ is roughly $\#\mathcal{X}^2$. One then merges the sets by finding matches $(\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}, \mathbf{x}_1, \mathbf{x}_2) \in L_1^{(0)}$ and $(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1}, \mathbf{x}_3, \mathbf{x}_4) \in L_2^{(0)}$ such that $\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 + \mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \equiv \mathbf{0} \pmod{H_1}$. Our observation is that we also have $(\psi^i(\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2) \pmod{H_1}, \psi^i(\mathbf{x}_1), \psi^i(\mathbf{x}_2)) \in L_1^{(0)}$ and $(\psi^i(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4) \pmod{H_1}, \psi^i(\mathbf{x}_3), \psi^i(\mathbf{x}_4)) \in L_2^{(0)}$, and these values satisfy

$$\psi^i(\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2) + \psi^i(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4) \equiv \mathbf{0} \pmod{H_1}.$$

Hence the idea is to store a single representative of each orbit. We now define the representative we use.

Definition 3. *Let notation be as above, with q prime and $2n \mid (q-1)$. Let $k = (q-1)/(2n)$ and let $\alpha_1, \dots, \alpha_k$ be coset representatives for $\mathbb{Z}_q^*/Z$ where $Z = \{z \in \mathbb{Z}_q^* : z^{2n} = 1\}$. Suppose ψ has distinct eigenvalues $\lambda_1, \dots, \lambda_n$ so that, with respect to the corresponding eigenbasis, $\psi((s_1, \dots, s_n)^T) = (s_1\lambda_1, \dots, s_n\lambda_n)^T$. Let $\boldsymbol{s} \in R_q/H$ for any subgroup H such that $\psi(H) = H$. Then the equivalence class representative $[\boldsymbol{s}]$ of $\boldsymbol{s}$ is defined to be the element of $\{\psi^i(\boldsymbol{s}) : 0 \le i < 2n\}$ such that $[\boldsymbol{s}] = (0, \dots, 0, \alpha_j, \star, \dots, \star)$ for some $1 \le j \le k$ and $\star$ denotes an arbitrary element.*

Note that $[\mathbf{s}] = (s_1\lambda_1^i, s_2\lambda_2^i, \dots, s_n\lambda_n^i)^T$ for some $0 \le i < 2n$. Writing $\lambda_j = \lambda_1^{a_j}$ one can write this as

$$(s_1\lambda_1^i, s_2\lambda_1^{a_2 i}, \dots, s_n\lambda_1^{a_n i})^T.$$

To compute the class representative efficiently one uses a precomputed table of discrete logarithms with respect to λ_1. In the case of SWIFFT we can take $\alpha_1 = 1$ and $\alpha_2 = 3$.

Returning to the CPW algorithm. The idea is to store a single representative of each class, so that the list $L_1^{(0)}$ becomes $\{([\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}], \mathbf{x}_1, \mathbf{x}_2) : \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}\}$. We store the specific pair $(\mathbf{x}_1, \mathbf{x}_2)$ such that $[\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}] = \mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}$. We now consider the merge operation. The question is how to find all tuples $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ such that

$$\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 + \mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \equiv \mathbf{0} \pmod{H_1}$$

when given only the classes $[\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}], [\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1}]$. In other words, we need to determine all indices i, i' such that

$$\psi^i(\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}) \equiv \psi^{i'}(-(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1})). \tag{7}$$

Lemma 4. *Suppose a quadruple $(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$ satisfies equation (7). Then $[\boldsymbol{a}_1\boldsymbol{x}_1 + \boldsymbol{a}_2\boldsymbol{x}_2 \pmod{H_1}] = [\boldsymbol{a}_3\boldsymbol{x}_3 + \boldsymbol{a}_4\boldsymbol{x}_4 \pmod{H_1}]$. Furthermore, if $[\boldsymbol{a}_1\boldsymbol{x}_1 + \boldsymbol{a}_2\boldsymbol{x}_2 \pmod{H_1}] = \boldsymbol{a}_1\boldsymbol{x}_1 + \boldsymbol{a}_2\boldsymbol{x}_2 \pmod{H_1}$ and $[\boldsymbol{a}_3\boldsymbol{x}_3 + \boldsymbol{a}_4\boldsymbol{x}_4 \pmod{H_1}] = \boldsymbol{a}_3\boldsymbol{x}_3 + \boldsymbol{a}_4\boldsymbol{x}_4 \pmod{H_1}$ then $i' = i$.*

Proof. Without loss of generality we can apply a suitable power of ψ to both sides, so that the left hand side equals $[\mathbf{a}_1\mathbf{x}_1 + \mathbf{a}_2\mathbf{x}_2 \pmod{H_1}]$, and so is a normalised vector $(0, \cdots, 0, \alpha_j, \star, \cdots)^T$. The right hand side is then $\psi^{i''}(-(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1}))$ for some integer i''. To get equality it follows that the right hand side is also a normalised vector, and so

$$\psi^{i''}(-(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1})) = [-(\mathbf{a}_3\mathbf{x}_3 + \mathbf{a}_4\mathbf{x}_4 \pmod{H_1})].$$

The result follows. □

It follows that all matches in the merge algorithm can be found by simply checking equality of equivalence class representatives. Hence, the merge algorithm is exactly the same as Algorithm 1.

It follows that the sizes of lists are reduced from 3^n to roughly $3^n/(2n)$ and that the merge time is reduced correspondingly. There are some additional challenges in programming the algorithm, but they should not lead to a slower program overall.

We now make a remark about success probability. The original CPW algorithm would have lists of size 3^n, and they would be merged by checking a condition in $G/H = \mathbb{Z}_q^\ell$. The heuristic argument is that the merged list would be of size roughly $(3^n)^2/q^\ell$, and hence consist of around $3^{2n}/(2nq^\ell)$ orbits under ψ. In our new algorithm we have lists of size $3^n/(2n)$ and, by Lemma 4, there is a unique match (namely, it is sufficient to test equality of the normalised class representatives as in Definition 3). Since the number of normalised class representatives is roughly $q^\ell/(2n)$ it follows that the size of the new list (whose entries are class representatives) should be approximately $(3^n/2n)^2/(q^\ell/2n) \approx 3^{2n}/(2nq^\ell)$, which is consistent with the original algorithm. This argument supports our claim in footnote 7 (Section 5.1) that the symmetries do not negatively impact the success probability of these algorithms when applied to structured matrices.

These ideas do not lead to a better attack on SWIFFT for two reasons:

1. It seems hard to use this idea for Ring-ISIS (and hence inverting SWIFFT) because ψ does not fix the inhomogenous term $\mathbf{s}$. Also, in that case we would not have the required condition $\psi(\mathcal{X}) = \mathcal{X}$. Instead, the main application seems to be finding collisions in SWIFFT by solving Ring-SIS.
2. We already have a speedup to the CPW algorithm by a factor of more than $2n$, by using the Hermite normal form and other ideas.

It is an open problem to combine the use of orbits under ψ with the Hermite normal form. We briefly explain the issue. Solving Ring-SIS is finding $\mathbf{x}_1, \dots, \mathbf{x}_{16}$ such that $\mathbf{0} = \mathbf{a}_1\mathbf{x}_1 + \cdots + \mathbf{a}_{16}\mathbf{x}_{16}$. Assuming $\mathbf{a}_{16} \in R_q$ is invertible, the Hermite normal form is

$$\mathbf{0} = (\mathbf{a}_{16}^{-1}\mathbf{a}_1)\mathbf{x}_1 + \cdots + (\mathbf{a}_{16}^{-1}\mathbf{a}_{15})\mathbf{x}_{15} \ + \ \mathbf{x}_{16}.$$

One then wants to perform an 8-list approximate-CPW algorithm on the sum of 15 elements. The problem is that 8 does not divide 15, so when we split the sum of 15 terms into 8 blocks, we are unable to get a clean division into a sum of ring elements. Of course, we can discard the ring structure and split the sum as we did in Section 5.1. But since blocks no longer correspond to ring elements, we do not have an action by ψ and hence do not seem to be able to achieve any speedup.

7 Experimental results

The purpose of this section is two-fold. First, to show that our time and space complexity estimates are robust: the actual running-time of the algorithm follows from the bit complexity (and hence size) estimate. Second, 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.

7.1 ISIS inversion $\mathcal{B} = \{0, 1\}$.

The parameters we used here are $n = 16, q = 11$ and m ranges from 96 to 160. 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 4 shows the running-time comparison of algorithms in five 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). Columns "#**E**" are the 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 (**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).[11] The notation ℓ_i denotes the constraints for each level in the computation; when there are 3

[11] This does not occur in Table 4, but we see it in Table 5. 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.

(respectively 4) values ℓ_i listed it means we are performing an 8-list (respectively 16-list) algorithm. Column "T. **E1/E2**" 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. Column "succ. **E1/E2**" denotes the success probability (at least one solution is found) by averaging over 5000 trials. It is noted that our improved algorithms have comparable success probability to their counter-parts.

Table 4. Comparison of algorithms for ISIS inversion $\mathcal{B} = \{0, 1\}$.

m	$\tilde{m}$ and ℓ_i **(E1)**	$\tilde{m}$ and ℓ_i **(E2)**	**#E1**	**#E2**	**T. E1/E2**	**succ. E1/E2**
96	96, (2, 5, 9)	96, (2, 5, 9)	17.08	14.08	99.87s/8.30s	99.6% / 99.6%
104	104, (3, 5, 8)	104, (4, 2, 10)	15.62	12.41	30.68s/3.38s	69.7% / 69.5%
112	112, (4, 4, 8)	112, (4, 4, 8)	14.49	12.00	24.63s/3.39s	90.8% / 90.8%
128	128, (4, 4, 8)	128, (1, 4, 2, 9)	14.70	11.57	15.78s/2.05s	63.5% / 63.2%
160	160, (2, 4, 4, 6)	160, (3, 2, 3, 8)	13.08	10.33	8.43s/1.36s	83.4% / 83.5%

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.

7.2 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 $\ell_1, \dots, \ell_t$ and then apply the 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 5 is analogous to that used in Table 4. Column "succ. **E3/E4**" also denotes the success probability by averaging over 5000 trials. By comparison, our improved algorithms take less time and have higher success probability than their counter-parts.

Table 5. Comparison of algorithms for ISIS inversion $\mathcal{B} = \{0, 1\}$.

m	$\tilde{m}$ and ℓ_i **(E3)**	$\tilde{m}$ and ℓ_i **(E4)**	**#E3**	**#E4**	**T. E3/E4**	**succ. E3/E4**
96	88, (2, 3, 4)	96, (3, 2, 3)	15.39	10.34	23.58s/1.43s	45.3 % / 55.1%
128	128, (1, 3, 3, 2)	96, (3, 2, 3)	13.55	10.34	15.68s/1.43s	15.8 % / 55.1%
144	144, (1, 4, 3, 2)	144, (2, 2, 2, 3)	13.91	10.29	16.21s/1.57s	83.5% / 96.9%
160	160, (2, 3, 3, 2)	160, (3, 1, 2, 3)	12.85	9.94	8.32s/1.62s	67.5% / 89.2%
176	176, (3, 2, 3, 2)	176, (1, 1, 2, 2, 3)	12.50	9.59	7.38s/1.46s	50.9% / 78.2%

8 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 that 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 (using HNF trick on the original ISIS) and obtains an improved algorithm to solve ISIS. We believe these ideas will be of interest to studying ISIS instances with low density and small coefficient sets.

Finally, Section 7 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 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. Matthieu Finiasz and Nicolas Sendrier, Security Bounds for the Design of Code-Based Cryptosystems, In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 88105. Springer, Heidelberg (2009)
11. 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.
12. Nick Howgrave-Graham and Antoine Joux, New Generic Algorithms for Hard Knapsacks, in H. Gilbert (ed.), EUROCRYPT 2010, Springer LNCS 6110 (2010) 235–256.
13. Nick Howgrave-Graham and Antoine Joux, New Generic Algorithms for Hard Knapsacks (preprint), 17 pages (undated). Available from `www.joux.biz/publications/Knapsacks.pdf`
14. 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.
15. Jeffrey C. Lagarias and Andrew M. Odlyzko, Solving low-density subset sum problems, J. ACM, 32(1):229-246, 1985.

16. Vadim Lyubashevsky, On Random High Density Subset Sums, Electronic Colloquium on Computational Complexity (ECCC) 007 (2005)
17. 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.
18. 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.
19. 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.
20. Lorenz Minder and Alistair Sinclair, The Extended k-tree Algorithm, J.Cryptol. 25 (2012) 349–382.
21. 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.
22. 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.
23. David Wagner, A Generalized Birthday Problem, in M. Yung (ed.), CRYPTO 2002, Springer LNCS 2442 (2002) 288–303.