PAPER *Special Section on Cryptography and Information Security*

Multiparallel MMT: Faster ISD Algorithm Solving High-Dimensional Syndrome Decoding Problem

Shintaro NARISADA[†a)], **Kazuhide FUKUSHIMA**[†], ***Members**, **and*** **Shinsaku KIYOMOTO**[†], ***Senior Member***

SUMMARY The hardness of the syndrome decoding problem (SDP) is the primary evidence for the security of code-based cryptosystems, which are one of the finalists in a project to standardize post-quantum cryptography conducted by the U.S. National Institute of Standards and Technology (NIST-PQC). Information set decoding (ISD) is a general term for algorithms that solve SDP efficiently. In this paper, we conducted a concrete analysis of the time complexity of the latest ISD algorithms under the limitation of memory using the syndrome decoding estimator proposed by Esser et al. As a result, we present that theoretically nonoptimal ISDs, such as May–Meurer–Thomae (MMT) and May–Ozerov, have lower time complexity than other ISDs in some actual SDP instances. Based on these facts, we further studied the possibility of multiple parallelization for these ISDs and proposed the first GPU algorithm for MMT, the multiparallel MMT algorithm. In the experiments, we show that the multiparallel MMT algorithm is faster than existing ISD algorithms. In addition, we report the first successful attempts to solve the 510-, 530-, 540- and 550-dimensional SDP instances in the Decoding Challenge contest using the multiparallel MMT.

key words: *syndrome decoding problem, code-based cryptography, information set decoding (ISD), graphics processing unit (GPU)*

1. Introduction

The security of code-based cryptosystems such as the McEliece cryptosystem is based on the syndrome decoding problem (SDP) [1]. Information set decoding (ISD) is known as a family of algorithms for efficiently solving SDP. ISD probabilistically finds a solution for a given SDP based on combinatorics. To date, many ISD algorithms have been proposed including Dumer [2], May–Meurer–Thomae (MMT) [3], Becker–Joux–May–Meurer (BJMM) [4], May–Ozerov (MO) [5] and Both–May (BM) [6]. In these papers, the asymptotic time complexity for each ISD was analyzed. For example, the asymptotic time complexity of Both–May in the setting of full distance decoding is $2^{0.0885n}$, which is known to be the smallest among existing ISDs. Here, n is the dimension of the given SDP. In addition to ISD, several papers provided a computational analysis of more general decoding techniques [7]–[9]. There is also work on an *estimator* that analyzes the actual time complexity of ISDs for SDPs with practical problem sizes associated with real cryptosystems [10]–[13].

To estimate a secure parameter set for code-based cryptosystems, it is important not only to verify the computational complexity of ISD algorithms theoretically but also to verify to what level of difficulty the actual SDP can be solved practically. There is existing research on fast ISD implementations, including FPGA [14] and GPU implementations of Dumer's algorithm [15]. Recently, Esser *et al.* presented fast CPU-based concrete implementations of the MMT and BJMM algorithms in [16]. There are also papers on proposals for *quantum* ISD algorithms, albeit simulation-based [17], [18]. However, these papers lacked a comparative study with other ISDs in terms of computational complexity.

1.1 Contributions

In this paper, we analyze the actual time complexity of major ISD algorithms under the limitation of memory using a syndrome decoding estimator [10]. We also derive the optimal parameters of each ISD algorithm applicable to real SDP instances, containing a difficulty level of approximately 2^{50}, which corresponds to the highest dimension of SDP actually solved up to now (August 2021). As a result, contrary to the existing asymptotic results, we confirmed that the MMT algorithm (asymptotic runtime: $2^{0.112n}$) is faster than BJMM ($2^{0.102n}$) for several memory sizes, including those suitable for current midrange PC/Servers. Additionally, as confirmed in [10], we showed that the May–Ozerov (MO) algorithm ($2^{0.953n}$) has a smaller runtime than Both–May (BM) ($2^{0.0885n}$) for some memory sizes and instances.

Furthermore, we found that the MMT algorithm can be massively parallelized without increasing the amount of memory required, and proposed a first GPU-optimized algorithm for MMT called Multiparallel MMT. In our experiments, we implemented Multiparallel MMT using the CUDA language and compared the runtime with those of existing CPU/GPU-based ISD algorithms. As a result, our proposed algorithm achieved relatively smaller expected runtime than conventional ISD for a 530-dimensional SDP instance. In addition, we conducted experiments on large-scale SDP instances in the Decoding Challenge [19], a cryptanalysis web contest regarding code-based cryptosystems, known as benchmark websites, to show the levels of difficulty of code-based cryptosystems that can actually be solved with the current algorithms and machine power. We succeeded in solving the 510-, 530-, 540- and 550-dimensional SDP for the first time using our proposed algorithm and several modern GPUs. We believe that our results will contribute to the selection of more rigorous security parameters for post-quantum cryptosystems.

Manuscript received March 15, 2022.
Manuscript revised July 19, 2022.
Manuscript publicized November 9, 2022.

[†]The authors are with KDDI Research, Inc., Fujimino-shi, 356-8502 Japan.
a) E-mail: sh-narisada@kddi-research.jp

DOI: 10.1587/transfun.2022CIP0023

2. Notation

Let $[i, j]$ be a set of integers $\{i, i+1, \ldots, j\}$. In particular, $[k] = \{1, \ldots, k\}$. An n-dimensional column vector is denoted by $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ and $\mathbf{x}[i] = x_i$. A subsequence of $\mathbf{x}$ from i to j $(i < j)$ is denoted by $\mathbf{x}_{[i,j]} = (x_i, \ldots, x_j)$. The zero vector is written as $\mathbf{0}$. A matrix of size $m \times n$ is denoted by $\mathbf{A} \in \mathbb{F}_2^{m \times n}$. $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}$. The horizontal concatenation of two matrices $\mathbf{A} \in \mathbb{F}_2^{m \times n}$ and $\mathbf{B} \in \mathbb{F}_2^{m \times n}$ is denoted by $(\mathbf{A} \mid \mathbf{B}) \in \mathbb{F}_2^{m \times 2n}$. A size of matrix $\mathbf{A} \in \mathbb{F}_2^{m \times n}$ is denoted by $|\mathbf{A}| = mn$. $\mathbf{I}$ is the the identity matrix, and $\mathbf{O}$ denotes the zero matrix. The Hamming weight for $\mathbf{x} \in \mathbb{F}_2^m$ is denoted by $\mathrm{wt}(\mathbf{x}) = |\{i \mid \mathbf{x}[i] = 1\}|$. The SDP is defined as follows:

Definition 1 (SDP)**.** For any integers n, k and w such that $k \leq n$ and $w \leq n$, consider a parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$ and a syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$. Find a vector (*solution*) $\mathbf{e} \in \mathbb{F}_2^n$ of $\mathrm{wt}(\mathbf{e}) = w$ such that $\mathbf{He} = \mathbf{s}$.

We denote a syndrome decoding problem with parameters n, k, w by $\mathrm{SDP}(n, k, w)$.

3. Information Set Decoding (ISD)

ISD is a general term for algorithms based on combinatorics to solve SDP efficiently. A variety of ISDs have been proposed thus far and can be roughly divided into two categories: algorithms without the nearest neighbor (NN) and those based on NN algorithms. In our paper, we deal with six major ISD algorithms: Prange [20], Dumer [2], May–Meurer–Thomae (MMT) [3], Becker–Joux–May–Meurer (BJMM) [4] for ISDs without NN, and May–Ozerov (MO) [5] and Both–May (BM) [6] for ISDs with NN.

Before explaining each ISD, we will describe the common framework of ISD algorithms. Inputs to the ISD are integers n, k, w, the parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$ and the syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$. ISD outputs $\mathbf{e} \in \mathbb{F}_2^n$ satisfying $\mathbf{He} = \mathbf{s}$ and $\mathrm{wt}(\mathbf{e}) = w$. ISD performs random column permutation and Gaussian elimination on the input $\mathbf{H}$ and $\mathbf{s}$. That is, for an invertible column permutation matrix $\mathbf{P} \in \mathbb{F}_2^{n \times n}$ and a matrix $\mathbf{G} \in \mathbb{F}_2^{(n-k) \times (n-k)}$ corresponding to the Gaussian elimination, let $(\mathbf{Q} \mid \mathbf{I}) \longleftarrow \mathbf{GHP}$ and $\hat{\mathbf{s}} \longleftarrow \mathbf{Gs}$. After this, a search algorithm is performed on the matrix $\mathbf{Q}$ to compute $\hat{\mathbf{e}} = \mathbf{Pe}$ satisfying $(\mathbf{Q} \mid \mathbf{I})\hat{\mathbf{e}} = \hat{\mathbf{s}}$ and $\mathrm{wt}(\hat{\mathbf{e}}) = w$. If such an $\hat{\mathbf{e}}$ is found, then $\mathbf{e} \longleftarrow \mathbf{P}^{-1}\hat{\mathbf{e}}$ is a solution of $\mathrm{SDP}(n, k, w)$. If $\hat{\mathbf{e}}$ is not found, then the above procedure is performed again for a different column permutation matrix $\mathbf{P}$. We show the common framework for ISD in Algorithm 1.

We will briefly describe the time complexity of ISD, also called the work factor (WF). Let T be the time complexity required for the single **for** loop (Lines 2–9) in Algorithm 1. Additionally, let P be the probability of successfully finding a solution with the single call of the search function (Line 6). In this case, the WF of an ISD is expressed as TP^{-1}. The actual values of T and P vary depending on the ISD algorithm.

Algorithm 1: ISD Framework

```
   Input: $n, k, w, \mathbf{H}, \mathbf{s}$
   Output: $\mathbf{e}$
1  $\mathbf{e} \leftarrow \perp$
2  for $i \leftarrow 1$ To $P^{-1}$ do
3      $\mathbf{P} \leftarrow$ pick one permutation randomly
4      $(\mathbf{Q} \mid \mathbf{I}) \leftarrow \mathbf{GHP}$
5      $\hat{\mathbf{s}} \leftarrow \mathbf{Gs}$
6      $\hat{\mathbf{e}} \leftarrow \mathrm{Search}(\mathbf{Q}, \hat{\mathbf{s}})$
7      if $\mathrm{wt}(\hat{\mathbf{e}}) = w$ then
8          $\mathbf{e} \leftarrow \mathbf{P}^{-1}\hat{\mathbf{e}}$
9      if $\mathbf{e} \neq \perp$ then break
10 return $\mathbf{e}$
```

3.1 Prange

The Prange algorithm is the first ISD proposed in 1962. In Prange, we do not perform any search on the matrix $\mathbf{Q}$ but only check the weights of $\hat{\mathbf{s}}$. If $\mathrm{wt}(\hat{\mathbf{s}}) = w$, then $\mathbf{e} = \mathbf{P}^{-1}\hat{\mathbf{e}}$ is a solution, where $\hat{\mathbf{e}} = (\mathbf{0}, \hat{\mathbf{s}})$. This is because if $\mathrm{wt}(\hat{\mathbf{s}}) = w$, then $(\mathbf{Q} \mid \mathbf{I})\hat{\mathbf{e}} = \hat{\mathbf{s}}$ and $\mathrm{wt}(\hat{\mathbf{e}}) = w$ are satisfied by taking $\hat{\mathbf{e}} = (\mathbf{0}, \hat{\mathbf{s}})$. Intuitively, it is the case that for a matrix $(\mathbf{Q} \mid \mathbf{I})$, all w columns corresponding to positions of "1"s in $\mathbf{e}$ are contained in the part of $\mathbf{I}$. Therefore, if a column permutation $\mathbf{P}$ is applied to $\mathbf{H}$ such that all w positions of "1"s in $\mathbf{e}$ are contained in $\mathbf{I}$, then Prange can find a solution. The probability of such a column permutation occurring is $P = \binom{n-k}{w} / \min(2^{n-k}, \binom{n}{w})$, and the expected number of loops required for Prange in Algorithm 1 is P^{-1}. The time complexity T required for one loop of Prange is the sum of the time complexity required for column permutation and Gaussian elimination, namely, $T_{\mathrm{ge}} = n(n-k)$. The space complexity S required for Prange is the size of the input matrix $\mathbf{H}$: $S = |\mathbf{H}| = n(n-k)$.

3.2 Dumer

In Dumer's algorithm, an input matrix $\mathbf{H}$ is transformed as follows:

$$\left(\begin{array}{c|c} \mathbf{Q}_1 & \mathbf{O} \\ \hline \mathbf{Q}_2 & \mathbf{I} \end{array}\right) \longleftarrow \mathbf{GHP}, \tag{1}$$

where $\mathbf{Q}_1 \in \mathbb{F}_2^{\ell \times (k+\ell)}$ and $\mathbf{Q}_2 \in \mathbb{F}_2^{(n-k-\ell) \times (k+\ell)}$ for an integer parameter $\ell > 0$. Then, we run the following search algorithm on the matrix $\mathbf{Q}_1$. First, we construct two lists L_1 and L_2 for the enumeration parameter $p > 0$:

$$L_1 = \{(\mathbf{e}_1, \mathbf{Q}_1\mathbf{e}_1) \mid \mathbf{e}_1 = (\mathbf{a}, \mathbf{0}), \mathbf{a} \in I\}, \tag{2}$$

$$L_2 = \left\{(\mathbf{e}_2, \mathbf{Q}_1\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell]}) \mid \mathbf{e}_2 = (\mathbf{0}, \mathbf{a}), \mathbf{a} \in I\right\}, \tag{3}$$

where we define a set of binary vectors of weight p: $I = \{\mathbf{a} \in \mathbb{F}_2^{(k+\ell)/2} \mid \mathrm{wt}(\mathbf{a}) = p\}$. L_1 and L_2 store the combination of p

columns chosen from the left and right halves of $\mathbf{Q}_1$, respectively. Then, for each element $(\mathbf{e}_1, \mathbf{x}_1) \in L_1$, we run a depth-1 search for an element $(\mathbf{e}_2, \mathbf{x}_2) \in L_2$ that satisfies $\mathbf{x}_1 = \mathbf{x}_2$. This search can be implemented using buckets in time $\max(|L_1|, |L_1|^2/2^\ell)$. For each pair satisfying $\mathbf{x}_1 = \mathbf{x}_2$, there exists a solution $\mathbf{e}$ if $\mathrm{wt}(\mathbf{Q}_2\mathbf{e}_1 + \mathbf{Q}_2\mathbf{e}_2 + \mathbf{s}_{[\ell+1,n-k]}) = w - 2p$, where $\mathbf{e} = \mathbf{P}^{-1}(\mathbf{e}_1 + \mathbf{e}_2, \mathbf{Q}_2\mathbf{e}_1 + \mathbf{Q}_2\mathbf{e}_2 + \mathbf{s}_{[\ell+1,n-k]})$. Here, we consider the success probability P of the column permutation $\mathbf{P}$ such that $\hat{\mathbf{e}} = \mathbf{P}\mathbf{e}$ has the correct form, i.e., w "1"s in $\hat{\mathbf{e}}$ are distributed as $p, p, w-2p$ in the coordinate intervals $[1, (k+\ell)/2], [(k+\ell)/2 + 1, k+\ell], [k+\ell+1, n]$ of $\hat{\mathbf{e}}$, respectively. By considering the distribution of the columns corresponding to the solutions, we obtain $P = \binom{(k+\ell)/2}{p}^2 \binom{n-k-\ell}{w-2p} / \min(2^{n-k}, \binom{n}{w})$. The time complexity T required for 1 loop of Dumer is the sum of the runtime required for column permutation and Gaussian elimination and the runtime required for Dumer's search (list construction and matching):

$$T = T_{\mathrm{ge}} + |L_1| + \max(|L_1|, |L_1|^2/2^\ell). \tag{4}$$

where $|L_1| = \binom{(k+\ell)/2}{p}$. The required memory is $S = |\mathbf{H}| + |L_1|$.

3.3 May–Meurer–Thomae

In the MMT algorithm, we consider the following transformation of the matrix $\mathbf{H}$:

$$\left(\begin{array}{c|c} \mathbf{Q}_1 & \\ \mathbf{Q}_2 & \mathbf{O} \\ \hline \mathbf{Q}_3 & \mathbf{I} \end{array}\right) \leftarrow \mathbf{GHP}, \tag{5}$$

where $\mathbf{Q}_1 \in \mathbb{F}_2^{\ell_1 \times (k+\ell)}$, $\mathbf{Q}_2 \in \mathbb{F}_2^{\ell_2 \times (k+\ell)}$ and $\mathbf{Q}_3 \in \mathbb{F}_2^{(n-k-\ell) \times (k+\ell)}$ for integer parameters $\ell_1 > 0$, $\ell_2 > 0$ and $\ell = \ell_1 + \ell_2$. First, we construct four depth-2 lists $L_{11}, L_{12}, L_{21}, L_{22}$ for the matrix $\mathbf{Q}_1$ as follows:

$$L_{11} = \{(\mathbf{e}_{11}, \mathbf{Q}_1\mathbf{e}_{11}) \mid \mathbf{e}_{11} = (\mathbf{a}, \mathbf{0}), \mathbf{a} \in I_2\}, \tag{6}$$

$$L_{12} = \{(\mathbf{e}_{12}, \mathbf{Q}_1\mathbf{e}_{12}) \mid \mathbf{e}_{12} = (\mathbf{0}, \mathbf{a}), \mathbf{a} \in I_2\}, \tag{7}$$

$$L_{21} = \{(\mathbf{e}_{21}, \mathbf{Q}_1\mathbf{e}_{21}) \mid \mathbf{e}_{21} = (\mathbf{a}, \mathbf{0}), \mathbf{a} \in I_2\}, \tag{8}$$

$$L_{22} = \left\{(\mathbf{e}_{22}, \mathbf{Q}_1\mathbf{e}_{22} + \hat{\mathbf{s}}_{[\ell_1]}) \mid \mathbf{e}_{22} = (\mathbf{0}, \mathbf{a}), \mathbf{a} \in I_2\right\}, \tag{9}$$

where $I_2 = \{\mathbf{a} \in \mathbb{F}_2^{(k+\ell)/2} \mid \mathrm{wt}(\mathbf{a}) = p/2\}$. Note that the weight of $\mathbf{a}$ is $p/2$, unlike in Dumer. Then, the MMT search starts from matching depth-2 lists regarding the ℓ_1-bit prefix of binary vectors. Namely, L_{11} is matched with L_{12}, and L_{21} is matched with L_{22}. For instance, for $(\mathbf{e}_{11}, \mathbf{Q}_1\mathbf{e}_{11}) \in L_{11}$, we run a depth-2 search for an element $(\mathbf{e}_{12}, \mathbf{Q}_1\mathbf{e}_{12}) \in L_{12}$ satisfying $\mathbf{Q}_1\mathbf{e}_{11} = \mathbf{Q}_1\mathbf{e}_{12}$. As a result, the following 2 depth-1 lists are obtained in time $\max(|L_{11}|, |L_{11}|^2/2^{\ell_1})$:

$$L_1 = \{(\mathbf{e}_1, \mathbf{Q}_2\mathbf{e}_1) \mid \mathrm{wt}(\mathbf{e}_1) = p, \mathbf{Q}_1\mathbf{e}_1 = \mathbf{0}\}, \tag{10}$$

$$L_2 = \left\{(\mathbf{e}_2, \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}) \mid \mathrm{wt}(\mathbf{e}_2) = p, \mathbf{Q}_1\mathbf{e}_2 = \hat{\mathbf{s}}_{[\ell_1]}\right\}, \tag{11}$$

where $\mathbf{e}_1 = \mathbf{e}_{11} + \mathbf{e}_{12}$ and $\mathbf{e}_2 = \mathbf{e}_{21} + \mathbf{e}_{22}$. Furthermore, for each depth-1 element $(\mathbf{e}_1, \mathbf{Q}_2\mathbf{e}_1) \in L_1$, MMT searches for an element $(\mathbf{e}_2, \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}) \in L_2$ satisfying $\mathbf{Q}_2\mathbf{e}_1 = \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}$ in time $\max(|L_1|, |L_1|^2/2^{\ell_2})$. In this way, we can compute the combination of $2p$ columns of $\mathbf{Q}$ whose $\ell = \ell_1 + \ell_2$ bit prefix exactly matches the syndrome $\mathbf{s}_{\hat{[\ell]}}$. Finally, the remaining $n-k-\ell$ rows of $\mathbf{Q}$ ($\mathbf{Q}_3$) are verified. For each pair of L_1 and L_2, we can derive the solution $\mathbf{e}$ if $\mathrm{wt}(\mathbf{Q}_3\mathbf{e}_1 + \mathbf{Q}_3\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell+1,n-k]}) = w - 2p$ as in Dumer's algorithm, where $\mathbf{e} = \mathbf{P}^{-1}(\mathbf{e}_1 + \mathbf{e}_2, \mathbf{Q}_3\mathbf{e}_1 + \mathbf{Q}_3\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell+1,n-k]})$. In this paper, the success probability P that $\hat{\mathbf{e}} = \mathbf{P}\mathbf{e}$ has the correct form by the random permutation is set to $P = \binom{(k+\ell)/2}{p}^2 \binom{n-k-\ell}{w-2p} / \min(2^{n-k}, \binom{n}{w})$ for simplicity as in the original paper [3], which is the same as Dumer. Note that the formula is not accurate since MMT can also find $\hat{\mathbf{e}}$ that "1"s in $\hat{\mathbf{e}}$ are distributed as, for example, $p-2, p, w-2p+2$ by considering the case where "$1+1=0$" that occurs when merging L_1 and L_2 as in BJMM. The time complexity required for 1 loop of MMT is the sum of the runtime required for the permutation, Gaussian elimination, and MMT search as follows:

$$T = T_{\mathrm{ge}} + |L_{11}| + T_1 + T_2, \tag{12}$$

where $|L_{11}| = \binom{(k+\ell)/2}{p/2}$ and $|L_1| = \max(1, |L_{11}|^2/2^{\ell_1})$. The time complexity of the depth-2 list matching $T_2 = \max(|L_{11}|, |L_{11}|^2/2^{\ell_1})$ and depth-1 list matching $T_1 = \max(|L_1|, |L_1|^2/2^{\ell_2})$. The space complexity of the MMT algorithm is $S = |\mathbf{H}| + |L_{11}| + |L_1|$.

Split Representations

For a vector $\mathbf{x} \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(\mathbf{x}) = w$, a split representation of $\mathbf{x}$ is a pair $(\mathbf{x}_1, \mathbf{x}_2)$ satisfying $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ and $\mathrm{wt}(\mathbf{x}_1) = \mathrm{wt}(\mathbf{x}_2) \geq w/2$. In the MMT algorithm, it is known that there are $\mathcal{R} = \binom{p}{p/2}^2$ split representations for each $\mathbf{x}$ if we consider the Cartesian product for depth-2 lists: $L_{11} \times L_{12} \times L_{21} \times L_{22}$. Since multiple representations are considered duplicates, we want to reduce the number of such representations until the number of representations for $\mathbf{x}$ is 1. To do so, in the phase where L_{11} and L_{12} are matched, filtering regarding the ℓ_1-prefix of $\mathbf{x}$ is applied. This reduces the number of representations to $\mathcal{RP}$, where $\mathcal{P}$ is the surviving probability of each representation by the filtering. In the MMT, $\mathcal{P} = 1/2^{\ell_1}$. In the original paper [3], the authors select the parameter ℓ_1 in the range $\mathcal{RP} \geq 1$ to set the number of representations to 1 or more. However, we can consider the case $\mathcal{RP} < 1$, which means stronger filtering. In this case, the success probability P' of obtaining a solution in one MMT search is $P' = P\mathcal{RP}$, where P is the probability of successful permutation. Together with $\mathcal{RP} \geq 1$, P' can be generalized to $P' = P\min(1, \mathcal{RP})$. Later we will discuss how strong the filtering rate should be to reduce the overall runtime of ISD for actual SDP instances.

3.4 Becker–Joux–May–Meurer

BJMM is a generalized algorithm of MMT in terms of the enumeration parameter p. BJMM can consider more numbers of split representations $\mathcal{R}$ than MMT. In other words, it is possible to apply stronger filtering while preserving the number of representations $\mathcal{RP} \geq 1$. In this paper, we consider the depth-2 BJMM algorithm, which has the same depth as MMT, for simplicity of description and implementation. BJMM differs from MMT in the construction of the depth-1 lists L_1 and L_2. In BJMM, the weights of $\mathbf{e}_1$ and $\mathbf{e}_2$ are $p_1 = p + 2\epsilon$ instead of p. To distinguish from MMT, we consider $\epsilon > 0$ hereafter. As a result, the weights of depth-2 lists L_{11}, L_{12}, L_{21}, and L_{22} become $p/2 + \epsilon$. When merging L_1 and L_2, BJMM generates $\mathbf{e} = \mathbf{e}_1 + \mathbf{e}_2$ with weight $2p$ from XORing of $\mathbf{e}_1$ and $\mathbf{e}_2$. That is, BJMM considers the case where the common 2ϵ 1s in $\mathbf{e}_1$ and $\mathbf{e}_2$ are cancelled out by $1 + 1 = 0$. In this case, the number of split representations is $\mathcal{R} = \binom{p}{p/2}^2 \binom{(k+\ell)/2-p}{\epsilon}^2$, which differs from MMT. The survival probability of representations is $\mathcal{P} = 1/2^{\ell_1}$, the same as for MMT. The success probability P is the same as MMT. The time complexity T is the case when $|L_{11}| = \binom{(k+\ell)/2}{p_1/2}$ in Eq. (12).

3.5 May–Ozerov

MO is an algorithm that introduces the nearest neighbor (NN) algorithm to ISD. In this subsection, we will explain the MO algorithm applied to the BJMM (or MMT). We set the parameter ℓ_1 used in the MMT to $\ell_1 = \ell$ ($\ell_2 = 0$). First, we construct four depth-2 lists $L_{11}, L_{12}, L_{21}, L_{22}$ and merge them into two depth-1 lists L_1, L_2 as in MMT. Then, L_1 and L_2 are merged using an NN algorithm. For the NN algorithm, we use the meet-in-the-middle (MITM) algorithm used in [6] and the locality sensitive hashing (LSH)-based algorithm proposed in [10]. The following theorem is known about the time complexity of these NN algorithms:

Theorem 1. (NN algorithm [6], [10]) For two lists L_1, L_2 of size L whose elements are vectors of length m, there exists an NN algorithm to find all pairs $(\mathbf{x}_1, \mathbf{x}_2) \in L_1 \times L_2$ whose Hamming distance is δ in time $T_N(L, m, \delta)$.

$$T_N(L, m, \delta) = \max\left(L\binom{m}{\delta/2}, L^2\binom{m}{\delta/2}^2/2^m\right) \text{ [6]}, \tag{13}$$

$$T_N(L, m, \delta) = \binom{m}{\delta}\Big/\binom{m-\lambda}{\delta}\max\left(L, L^2/2^\lambda\right) \text{ [10]}, \tag{14}$$

where $\lambda = \min(\lg L, m - 2\delta)$.

Note that an NN algorithm [6] finds all pairs exactly, but another NN algorithm [10] does not necessarily find all solutions since it solves the problem probabilistically by using the locality sensitive hashing technique. In the merging of L_1 and L_2, the NN algorithm with weight $w - 2p$ is performed on a vector of length $n - k - \ell$. Thus, the time complexity of the NN algorithm is $T_N(|L_1|, n - k - \ell, w - 2p)$ from Theorem 1. A solution of the NN algorithm is the solution of the SDP. The success probability P that $\hat{\mathbf{e}} = \mathbf{Pe}$ has the correct form by the random permutation is $P = \binom{(k+\ell)/2}{p}^2 \binom{n-k-\ell}{w-2p}/\min(2^{n-k}, \binom{n}{w})$ in MO, which is same as MMT and BJMM. The time complexity of MO is as follows:

$$T = T_{\mathrm{ge}} + |L_{11}| + T_1 + T_2, \tag{15}$$

where $|L_{11}|$ and T_2 are the same as those of BJMM, and $T_1 = T_N(|L_1|, n - k - \ell, w - 2p)$. The required memory $S = T_{\mathrm{ge}} + |L_{11}| + |L_1|$, number of split representations $\mathcal{R}$ and survival probability $\mathcal{P}$ are the same as in BJMM.

3.6 Both–May

BM is an ISD proposed in 2018 based on the NN algorithm following MO. This paper addresses the depth-2 BM for comparison with other ISDs. The input matrix $\mathbf{H}$ is transformed into

$$\left(\begin{array}{c|c} \mathbf{Q}_1 & \multirow{2}{*}{$\mathbf{I}$} \\ \mathbf{Q}_2 & \end{array}\right) \leftarrow \mathbf{GHP}, \tag{16}$$

where $\mathbf{Q}_1 \in \mathbb{F}_2^{\ell \times k}$ and $\mathbf{Q}_2 \in \mathbb{F}_2^{(n-k-\ell)\times k}$ for a parameter ℓ. First, we construct four lists L_{12}, L_{22}, L_{21} and L_{22} from the matrix $\mathbf{Q}_1$ similar to MMT:

$$L_{11} = \{(\mathbf{e}_{11}, \mathbf{Q}_1\mathbf{e}_{11})) \mid \mathbf{e}_{11} = (\mathbf{a}, \mathbf{0}), \mathbf{a} \in I_2\}, \tag{17}$$

$$L_{12} = \{(\mathbf{e}_{12}, \mathbf{Q}_1\mathbf{e}_{12})) \mid \mathbf{e}_{12} = (\mathbf{0}, \mathbf{a}), \mathbf{a} \in I_2\}, \tag{18}$$

$$L_{21} = \{(\mathbf{e}_{21}, \mathbf{Q}_1\mathbf{e}_{21})) \mid \mathbf{e}_{21} = (\mathbf{a}, \mathbf{0}), \mathbf{a} \in I_2\}, \tag{19}$$

$$L_{22} = \left\{(\mathbf{e}_{22}, \mathbf{Q}_1\mathbf{e}_{22} + \mathbf{s}_{[\ell]}))\mid \mathbf{e}_{22} = (\mathbf{0}, \mathbf{a}), \mathbf{a} \in I_2\right\} \tag{20}$$

where $I_2 = \{\mathbf{a} \in \mathbb{F}_2^{k/2} \mid \mathrm{wt}(\mathbf{a}) = p_1/2\}$ and $p_1 = p + 2\epsilon$. In BM, we assume $\epsilon \geq 0$ in order to consider both MMT and BJMM cases. Then, these four lists are merged into two lists L_1 and L_2 using an NN algorithm:

$$L_1 = \{(\mathbf{e}_1, \mathbf{Q}_2\mathbf{e}_1) \mid \mathrm{wt}(\mathbf{e}_1) = p_1, \mathrm{wt}(\mathbf{Q}_1\mathbf{e}_1) = w_1\}, \tag{21}$$

$$\begin{aligned} L_2 = &\{(\mathbf{e}_2, \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell+1,n-k]}) \mid \\ &\quad \mathrm{wt}(\mathbf{e}_2) = p_1, \mathrm{wt}(\mathbf{Q}_1\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell]}) = w_1\}, \end{aligned} \tag{22}$$

where $w_1 \geq 0$ is a weight parameter, $\mathbf{e}_1 = \mathbf{e}_{11} + \mathbf{e}_{12}$ and $\mathbf{e}_2 = \mathbf{e}_{21} + \mathbf{e}_{22}$. The time complexity of the depth-2 NN is $T_N(|L_{11}|, \ell, w_1)$ from Theorem 1. Depth-1 lists L_1 and L_2, which are merged using an NN algorithm as well as MO. In particular, consider the weight parameter $w_2 \leq 2w_1$ and perform an NN algorithm with length $n - k - \ell$ and weight $w - 2p - w_2$. This yields the following set $\mathcal{I}$:

$$\mathcal{I} = \{(\mathbf{e}', \mathbf{e}'') \mid \mathrm{wt}(\mathbf{e}') = 2p, \mathrm{wt}(\mathbf{e}'') = w - 2p\}, \tag{23}$$

where $\mathbf{e}' = \mathbf{e}_1 + \mathbf{e}_2$, $\mathrm{wt}(\mathbf{Q}_2\mathbf{e}') = w - 2p - w_2$, $\mathrm{wt}(\mathbf{Q}_1\mathbf{e}') = w_2$, and $\mathbf{e}'' = (\mathbf{Q}_1\mathbf{e}', \mathbf{Q}_2\mathbf{e}')$. The time complexity of the NN is $T_N(|L_1|, n - k - \ell, w - 2p - w_2)$. If $\mathcal{I} \neq \emptyset$, then any element of $\mathcal{I}$ is a solution of SDP. The success probability of the BM is $P = \binom{k/2}{p}^2 \binom{\ell}{w_2} \binom{n-k-\ell}{w-2p-w_2}/\min(2^{n-k}, \binom{n}{w})$. The time complexity for one BM search T is the case when $|L_{11}| = \binom{k/2}{p_1}$, $|L_1| = \max(1, |L_{11}|^2\binom{\ell}{w_1}/2^\ell)$, $T_2 = T_N(|L_{11}|, \ell, w_1)$, $T_1 = T_N(|L_1|, n - k - \ell, w - 2p - w_2)$ in Eq. (12). The

Table 1 Asymptotic runtime for each ISD (value α for $2^{\alpha n}$).

Algorithm	Prange	Dumer	MMT	BJMM	MO	BM
Asymptotic runtime	0.121	0.117	0.112	0.102	0.0953	0.0885

Table 2 Optimal complexity for SDP(500, 250, 61).

Alg.	$\lg \mathrm{WF}$	$\lg T$	$-\lg P$	$\lg S$	$\lg \mathcal{RP}$
Prange	70.62	16.93	53.68	16.93	–
Dumer	53.63	20.07	33.55	20.07	–
MMT	51.99	29.79	21.84	28.79	−0.36
BJMM	51.33	38.83	11.90	37.64	−0.61
MO	50.95	40.67	9.58	30.49	−0.69
BM	52.04	40.33	6.02	36.23	−5.69

Table 3 Optimal parameters for SDP(500, 250, 61).

Algorithm	p	e	ℓ	ℓ_1	ℓ_2	w_1	w_2
Dumer	3	–	18	–	–	–	–
MMT	6	–	38	9	29	–	–
BJMM	10	1	68	31	37	–	–
MO	8	1	27	–	–	–	–
BM	14	2	48	–	–	1	2

required memory S and number of representations $\mathcal{R}$ are the same as with BJMM. In summary, the filtering part of the BM corresponds to the constraint regarding weights w_1 and w_2 for vectors of length ℓ. The survival probability $\mathcal{P}$ is $\mathcal{P} = \binom{w_2}{w_2/2}\binom{\ell-w_2}{w_1-w_2/2}/2^{\ell}$.

4. Complexity Analysis

In this section, we briefly describe the syndrome decoding estimator (SDE) and analyze the actual time complexity for each ISD under the memory constraint. SDE refers to a program that calculates the optimal time/space complexity and its parameters of ISDs for *actual* SDP instances. While there are several existing studies on SDE [10], [12], [13], we use an SDE proposed by Esser and Bellini [10].

First, we show the *asymptotic* time complexity for each ISD in the full distance decoding setting in Table 1. Full distance decoding is a problem setting of SDP where input parameters satisfy $\binom{n}{w} \approx 2^{n-k}$, where the values in Table 1 correspond to α in the asymptotic runtime $2^{\alpha n}$ for the full distance decoding that is maximized over all constant $0 \leq c \leq 1$ with the asymptotic parameter $k = cn$. From Table 1, BM has the smallest asymptotic runtime.

4.1 Syndrome Decoding Estimator (SDE)

An SDE takes $\mathrm{SDP}(n, k, w)$ as input and outputs the optimal work factor $\mathrm{WF} = TP^{-1}$, runtime required for one search call T, success probability P, required memory S and parameters for these complexities. The following is the concrete procedure of the SDE to compute the optimal work factor. First, for $\mathrm{SDP}(n, k, w)$, the SDE computes the set of feasible integer parameters $\mathcal{J}$ for each ISD. Then, while computing T, P, S and WF for each $j \in \mathcal{J}$, we keep the minimum work factor WF and parameter $j_{\min}$ at that time. The SDE outputs WF and $j_{\min}$ stored after processing for all feasible parameter sets as optimal values. The formulae for T, P and S of each ISD are given in Sect. 3.

4.2 Optimal WF and Parameters for Each ISD

In this subsection, we present the results of the analysis using Esser's SDE for several SDP instances. First, we show the optimal time complexity of each ISD when memory is unlimited. As an actual SDP instance, we consider $\mathrm{SDP}(n = 500, k = 250, w = 61)$. This SDP has a difficulty of approximately 2^{53} in the Decoding Challenge [19] and was the most difficult SDP that had been successfully solved as of August 2021. We show the optimal WF and parameters of each ISD calculated by the SDE in Table 2. Note that $\log_2$ is applied to all values. First, with respect to WF, a comparison of the asymptotic runtime in Table 1 and actual runtime in Table 2 indicates that BM has a larger WF than other ISDs such as MMT, BJMM and MO, unlike the results for the asymptotic analysis. For T and P^{-1} (the number of loops), T tends to be larger and P^{-1} smaller as one moves down Table 1. Thus, the newer ISDs tend to have a higher search cost and higher success probability for one search when WF is minimized.

The required memory S is proportional to the search cost T. It can also be seen that BJMM consumes the most memory under the optimal WF. The relationship between S and the actual memory usage depends on the size of the data type that stores one element. For example, if one uses an 8-byte long long type, then at least $8S$ bytes of memory is required in total. In this paper, we assume that the 8-byte data type is used. For the expected number of representations after filtering $\mathcal{RP}$, all ISDs using the representation have the smallest WF when $\lg \mathcal{RP} < 0$. That is, considering the case $\lg \mathcal{RP} < 0$, which implies stronger filtering, helps to reduce the WF of the ISD. The parameters of each ISD that derive the optimal WF are listed in Table 3. Prange is omitted because it has no parameter.

4.3 Optimal WF under Memory Constraints

Next, we analyze how the computational complexity of each ISD changes when the amount of available memory is constrained. We calculated the optimal WF for each ISD under memory constraints for $\mathrm{SDP}(500, 250, 61)$ and a much more difficult instance: $\mathrm{SDP}(1000, 500, 119)$. The results are shown in Fig. 1 and Fig. 2. The x-axis represents the maximum amount of memory that can be used. The y-axis represents WF. The optimal WF and S of Prange are $\lg \mathrm{WF} = 70.62$ and $\lg S = 16.93$ for $\mathrm{SDP}(500, 250, 61)$ and $\lg \mathrm{WF} = 127.66$ and $\lg S = 18.93$ for $\mathrm{SDP}(1000, 500, 119)$. First, we compare three non-NN-based methods: Dumer, MMT, and BJMM. Dumer has the largest WF for most amounts of memory, but the memory requirement for the optimal WF is small. For MMT and BJMM, contrary to the asymptotic analysis, the WF of MMT is smaller than that of BJMM in the range $\lg S < 31$ for SDP(500, 250, 61) and $\lg S < 41$ for SDP(1000, 500, 119). Note that the asymptotic runtime of BJMM could be larger than MMT since the asymptotic result does not consider the memory limitation. To solve SDP(500, 250, 61), MMT seems to be theoretically faster than BJMM when using midrange workstations or GPUs since 2^{31} corresponds to 8 GB, and it requires several times as much memory as 8 GB practically. For the two NN-based methods, MO was found to be faster than BM for any S, contrary to the asymptotic result. Consequently, MO has the smallest WF.

Comparing Fig. 1 and Fig. 2, it can be seen that the amount of memory required for the optimal WF increases drastically for larger SDPs. However, it is physically impossible to allocate 2^{90} of memory to achieve the optimal WF of MO in SDP(1000, 500, 119) in practice. Assuming a midrange computer is used, it is realistic to choose an ISD near $S \sim 2^{27}$. Looking at the area around $\lg S \approx 27$ in Fig. 1, we can see that the WFs of MO and MMT are relatively small. MO has a smaller WF than MMT in both figures. However, we emphasize that MO has larger polynomial factors than MMT. Therefore, we focus on the MMT to propose a multiparallel algorithm in this paper.

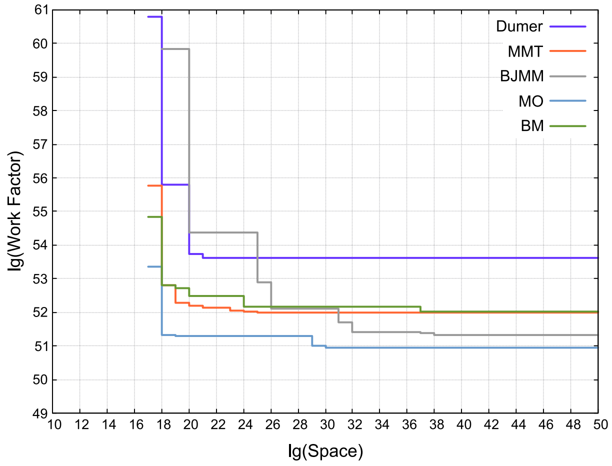

Fig. 1 Work factor under memory constraint for SDP(500, 250, 61).

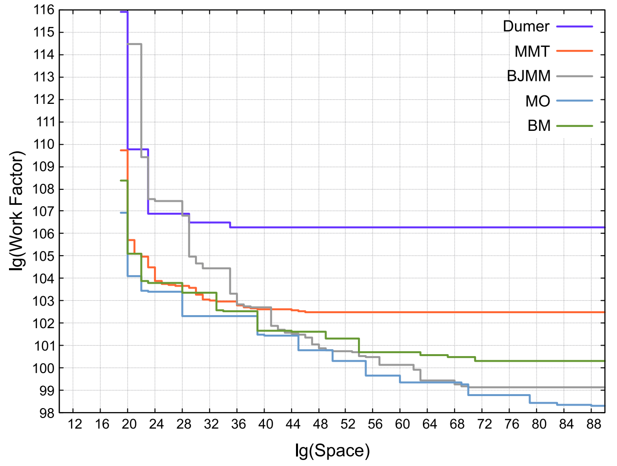

Fig. 2 Work factor under memory constraint for SDP(1000, 500, 119).

5. Parallel Optimization for MMT

We presented that the runtime of MMT is relatively small compared to other ISDs when the memory usage is up to several GB. In this section, we construct a fast algorithm for MMT using the parallelization technique and propose the first multiparallel optimization algorithm for MMT, the multiparallel MMT.

5.1 Multiparallel MMT Algorithm

First, the runtime of MMT for each process (corresponding to Eq. (12)) was extracted under the optimal WF to identify the process that should be parallelized. The results are listed in Table 4. Here, the target SDP instance is SDP(500, 250, 61). We choose $p = 2, 4, 6$, which corresponds to the memory sizes of small, medium and large, respectively. Especially, the parameter setting $p = 6$ produces the optimal WF among all p. For example, the optimal WF is 52.43 with a memory size of 35.46 in the case where $p = 8$. From Table 4, T_2 and T_1 seem to be bottlenecks of entire runtime for $p = 4, 6$ and T_{ge} and T_1 are bottlenecks for $p = 2$. In fact, runtime profiling using GNU Profiler showed that the actual runtime of T_2, T_1 accounted for more than 95% of the total for $p = 4, 6$, and Gaussian elimination accounted for 54% for $p = 2$ when using the optimal parameters obtained by the SDE. Therefore, we decided to adopt the following strategy in parallelizing MMT.

- We focus on $p = 4, p = 6$ since WF is smaller than $p = 2$.
- We use GPU to accelerate the heavy processing corresponding to T_2 and T_1, namely, list merging for depth-2 and depth-1.

We describe in detail the multiparallel construction and merging of lists, which is the core of the multiparallel MMT.

Multiparallel Construction and Merging

In multiparallel MMT, a depth-2 list L_{11} is first constructed on the CPU as a two-dimensional static array $\mathcal{L}_{11}$. This is because the optimal runtime $|L_{11}|$ in Table 4 is sufficiently small compared to other processes. Note that we do not actually need to construct L_{12}, L_{21} and L_{22} as already suggested in [21]. $\mathcal{L}_{11}[i][j]$ stores the j-th element $\mathbf{e}_{11}$ such that $i = \mathbf{Q}_1\mathbf{e}_{11}$. We set $0 \leq i < 2^{\ell_1}$ and

Table 4 Runtime of MMT for each process at $p = 2, 4, 6$.

$\lg S$	p	$\lg \mathrm{WF}$	$\lg T_{\mathrm{ge}}$	$\lg \lvert L_{11} \rvert$	$\lg T_2$	$\lg T_1$
16.98	2	55.78	16.93	7.02	12.04	16.09
20.55	4	52.13	16.93	13.21	20.41	20.83
28.79	6	51.99	16.93	18.89	28.79	28.79

$0 \leq j < \binom{(k+\ell)/2}{p/2}/2^{\ell_1}$ since there are 2^{ℓ_1} distinct vectors $\mathbf{Q}_1\mathbf{e}_{11}$ and the expected number of $\mathbf{Q}_1\mathbf{e}_{11}$ stored in the i-th bucket $\mathcal{L}_{11}[i]$ is $\binom{(k+\ell)/2}{p/2}/2^{\ell_1}$. $\mathcal{L}_{11}$ can be implemented by a static integer array using simple integer hashing technique. See the reference implementation in Appendix for details. Of course, the static bucket size will cause leakage when the actual number of elements to be stored in bucket $\mathcal{L}_{11}[i]$ exceeds $\binom{(k+\ell)/2}{p/2}/2^{\ell_1}$, but to simplify the process, we ignore this case. $\mathcal{L}_{11}$ is copied to the global memory in the GPU after construction.

Next, we explain the depth-2 merging on the GPU. In multiparallel MMT, $|L_{12}| = \binom{(k+\ell)/2}{p/2}$ elements are enumerated by threads on GPU in parallel instead of actually constructing L_{12}, and each thread searches matched elements in $\mathcal{L}_{11}$ in parallel to generate the depth-1 list L_1. Namely, each thread storing one pair $(\mathbf{e}_{12}, \mathbf{Q}_1\mathbf{e}_{12}) \in L_{12}$ accesses $\mathcal{L}_{11}$ by $i = \mathbf{Q}_1\mathbf{e}_{12}$. If $\mathcal{L}_{11}[i]$ is not empty, we construct a pair $(\mathbf{e}_{11}, \mathbf{e}_{12})$ for each element $\mathbf{e}_{11} \in \mathcal{L}_{11}[i]$ and $\mathbf{e}_{12}$. For such a pair, $(\mathbf{e}_{11} + \mathbf{e}_{12}, \mathbf{Q}_2(\mathbf{e}_{11} + \mathbf{e}_{12})) \in L_1$. Actually, L_1 is constructed in parallel as a one-dimensional static array $\mathcal{L}_1$ on the GPU. $\mathcal{L}_1[i]$ stores $\mathbf{e}_{11} + \mathbf{e}_{12}$ for $i = \mathbf{Q}_2(\mathbf{e}_{11} + \mathbf{e}_{12})$, where $0 \leq i < \ell_2$. That is, we set the bucket size of $\mathcal{L}_1[i]$ to 1 since we found that $|L_1| \sim 2^{\ell_2}$ is satisfied when WF is optimized. Therefore, loss of candidates may occur in $\mathcal{L}_1$ as similar as in $\mathcal{L}_{11}$. We will discuss the effect of the leakage on the success probability P in the next subsection. Since all threads access $\mathcal{L}_1$ at the same time, concurrent writes to the same location i may occur. We handle this by applying competitive writing. Competitive writing is a write without thread synchronization where only one thread is guaranteed to write to the location. Since $\mathcal{L}_1[i]$ can store only one element, we do not need to use thread synchronization. Similarly, $\mathcal{L}_2$ is constructed from $\mathcal{L}_{11}$.

$\mathcal{L}_1$ and $\mathcal{L}_2$ are merged using parallel processing on the GPU. All the 2^{ℓ_2} elements in $\mathcal{L}_2$ are enumerated in parallel. Then, each thread searches the pair from $\mathcal{L}_1$ in parallel. Each thread having $(\mathbf{e}_2, \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]})$ accesses $\mathcal{L}_1[i]$ by $i \leftarrow \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}$. Then, if $\mathcal{L}_1[i] \neq \emptyset$, $(\mathbf{e}_1, \mathbf{e}_2)$ is a matched pair, where $\mathbf{e}_1 \leftarrow \mathcal{L}_1[i]$. Finally, each thread checks whether the Hamming distance between $\mathbf{Q}_3(\mathbf{e}_1 + \mathbf{e}_2)$ and $\hat{\mathbf{s}}_{[\ell+1,n-k]}$ is exactly $w - 2p$. If it passes, $\mathbf{e} \leftarrow \mathbf{P}^{-1}(\mathbf{e}_1 + \mathbf{e}_2, \mathbf{Q}_3\mathbf{e}_1 + \mathbf{Q}_3\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell+1,n-k]})$ is a solution of a given SDP.

We summed up the above procedure in Algorithm 2. Algorithm 2 is repeatedly called as the search function in Algorithm 1 until a solution $\mathbf{e}$ is successfully found. The major difference between our previous GPU implementation [15] and the implementation of our proposed algorithm is thread synchronization when creating the depth-1 list L_1. In the previous implementation, L_1 is a one-dimensional list constructed by synchronously counting the number of elements in each bucket using the CUDA `atomicAdd` function without losing candidates. Unlike Dumer, the size of L_1 is not fixed in MMT since L_1 is constructed by merging L_{11} and L_{12}. We cannot apply the synchronous counting in the multiparallel Dumer to the multiparallel MMT since dynamic memory allocation is needed to construct L_1 for MMT. Our new implementation constructs L_1 without thread synchronization, which incurs the loss of candidates. We experimentally confirmed that the candidate loss has little impact on the runtime.

Algorithm 2: Search of multiparallel MMT

```
Input: $\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{Q}_3, \hat{\mathbf{s}}$
Output: $\mathbf{e}$
$\mathbf{e} \leftarrow \perp$
Construct $\mathcal{L}_{11}$ from $\mathbf{Q}_1$
Initialize $\mathcal{L}_1$, $\mathcal{L}_2$
parallel for $(\mathbf{e}_{12}, \mathbf{Q}_1\mathbf{e}_{12}) \in L_{12}$ do
    $i \leftarrow \mathbf{Q}_1\mathbf{e}_{12}$
    for $\mathbf{e}_{11} \in \mathcal{L}_{11}[i]$ do
        $x \leftarrow \mathbf{Q}_2(\mathbf{e}_{11} + \mathbf{e}_{12})$
        $\mathcal{L}_1[x] \leftarrow \mathbf{e}_{11} + \mathbf{e}_{12}$
    $j \leftarrow \mathbf{Q}_1\mathbf{e}_{12} + \hat{\mathbf{s}}_{[\ell_1]}$
    for $\mathbf{e}_{11} \in \mathcal{L}_{11}[j]$ do
        $x \leftarrow \mathbf{Q}_2(\mathbf{e}_{11} + \mathbf{e}_{12}) + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}$
        $\mathcal{L}_2[x] \leftarrow \mathbf{e}_{11} + \mathbf{e}_{12}$
parallel for $\mathbf{e}_2 \in \mathcal{L}_2$ do
    $i \leftarrow \mathbf{Q}_2\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell_1+1,\ell]}$
    $\mathbf{e}_1 \leftarrow \mathcal{L}_1[i]$
    $x \leftarrow \mathbf{Q}_3(\mathbf{e}_1 + \mathbf{e}_2) + \hat{\mathbf{s}}_{[\ell+1,n-k]}$
    if $\mathrm{wt}(x) = w - 2p$ then
        $\mathbf{e} \leftarrow \mathbf{P}^{-1}(\mathbf{e}_1, \mathbf{e}_2, \mathbf{Q}_3\mathbf{e}_1 + \mathbf{Q}_3\mathbf{e}_2 + \hat{\mathbf{s}}_{[\ell+1,n-k]})$
return $\mathbf{e}$
```

5.2 Complexity Analysis

We analyze the complexity of the multiparallel MMT algorithm. Replacing the search function of Algorithm 1 with Algorithm 2, the time complexity T of the multiparallel MMT required for one loop in Algorithm 1 is

$$T = T_{\mathrm{ge}} + |L_{11}| + N^{-1}(T_1 + T_2), \tag{24}$$

where N is the number of threads. Note that the number of threads on a GPU cannot be assumed to be N since the per-thread performance of a GPU is generally lower than that of a CPU. For this reason, T in the multiparallel MMT is measured as the actual processing time. The success probability P' of obtaining a solution in one search of the multiparallel MMT is $P' = P\min(1, \mathcal{R}\mathcal{P}P_{L_{11}}P_{L_1})$, where P, $\mathcal{R}$ and $\mathcal{P}$ are the same as in the MMT. $P_{L_{11}}$ and P_{L_1} are the survival probabilities of each candidate in constructing lists $\mathcal{L}_{11}$ and $\mathcal{L}_1$, respectively. $P_{L_{11}}$ and P_{L_1} are approximately obtained as follows:

$$P_{L_{11}} = \frac{1}{2^{\ell_1}} + \frac{1}{|L_{11}|}\sum_{i=|L_{11}|/2^{\ell_1}+1}^{|L_{11}|} p(i) \tag{25}$$

$$P_{L_1} = \frac{1}{|L_1|}\sum_{i=1}^{|L_1|} q(i), \tag{26}$$

where $p(i) = \sum_{c=0}^{|L_{11}|/2^{\ell_1}-1}\binom{i-1}{c}(1/2^{\ell_1})^c(1-1/2^{\ell_1})^{i-1-c}$. and $q(i) = \left(\frac{2^{\ell_2}-1}{2^{\ell_2}}\right)^{|L_1|-i}$. Considering $P_{L_{11}}$, we assume that when

Table 5 Effect of number of threads on runtime for SDP(550, 275, 67).

#threads for $\mathcal{L}_{11}$	16	128	1024	11175
#threads for $\mathcal{L}_2$	16	128	1024	262144
T (ms)	3005.55	402.37	57.14	1.63
Ratio	1840.40	246.38	34.99	–

Table 6 Runtime comparison for SDP(250, 125, 32).

p	Algorithm	Runtime	Ratio
4	MMT	22.94s	20.67
	Multi-Parallel MMT	1.11s	–
6	MMT	108.10s	51.95
	Multi-Parallel MMT	2.11s	–

the i-th element $\mathbf{e}_{11}^{(i)}$ is attempted to be written to $\mathcal{L}_{11}[\mathbf{x}_i]$ by $\mathbf{x}_i = \mathbf{Q}_3\mathbf{e}_{11}^{(i)}$, processing of up to $(i-1)$-elements has been completed. Then $p(i)$ represents the probability that $c < B$ out of $i-1$ pieces has been placed in the bucket $\mathcal{L}_{11}[\mathbf{x}_i]$ of size $B = |L_{11}|/2^{\ell_1}$. That is, $p(i)$ is the probability that there is one or more vacancies in bucket $\mathcal{L}_{11}[\mathbf{x}_i]$. $P_{L_{11}}$ is the sum of $p(i)$ for all i. For P_{L_1}, we assume that i-th element $\mathbf{e}_1^{(i)}$ is successfully written to $\mathcal{L}_1[\mathbf{x}_i]$ by $\mathbf{x}_i = \mathbf{Q}_2\mathbf{e}_1^{(i)}$. Then, the remaining $|L_1| - i$ elements are randomly overwritten into 2^{ℓ_2} buckets of $\mathcal{L}_1$. $q(i)$ denotes the probability that the remaining $|L_1| - i$ pieces are written outside of bucket $\mathcal{L}_1[\mathbf{x}_i]$ and $\mathbf{e}_1^{(i)}$ remains in $\mathcal{L}_1$ until the end. Additionally, we rewrite $P_{L_1} = \frac{2^{\ell_2}}{|L_1|}(1 - \exp^{-|L_1|/2^{\ell_2}})$ by approximation.

5.3 Experimental Results

We show the experimental results of the multiparallel MMT. We implemented our algorithm using `C++17` and `CUDA 11.0`. We used an AMD Ryzen 9 3900 for CPU and an NVIDIA Tesla V100 for a GPU. First, we compared the runtime of the ordinary CPU-based MMT and multiparallel MMT for $p = 4, 6$. We used SDP(250, 125, 32) as a solvable SDP instance. For both MMT and multiparallel MMT, we set the parameters $\ell_1 = 6$ and $\ell_2 = 18$ for $p = 4$ and $\ell_1 = 6$ and $\ell_2 = 21$ for $p = 6$. The random seed for the column permutation was fixed. Table 6 lists the results. Ratio means the runtime of MMT divided by the that of multiparallel MMT. From Table 6, it is confirmed that our proposed algorithm solved the SDP several tens of times faster than the normal MMT for both parameters $p = 4, 6$.

We conducted an ablation study on the impact of the number of threads N on runtime. For SDP(550, 275, 67), we measured the runtime for one loop in Algorithm 1 when varying the number of GPU threads. The optimal parameters $p = 4, \ell_1 = 8, \ell_2 = 18$ were used. The results are shown in Table 5. #threads for $\mathcal{L}_{11}$ in Table 5 (resp. #threads for $\mathcal{L}_2$) corresponds to the number of threads in Line 4 (resp. Line 13) in Algorithm 2. The maximum number of threads in Line 4 is $|\mathcal{L}_{11}| = \binom{(k+\ell)/2}{p/2} = 11175$ and $|\mathcal{L}_2| = 2^{\ell_2} = 262144$ in Line 13. Table 5 shows that T decreases as the number of threads increases and supports the correctness of Eq. (24) of the multiparallel MMT.

Table 7 Expected runtime for SDP(550, 275, 67).

Alg.	T(ms)	$-\lg P$	Runtime	Ratio
MMT	522	34.15	872.13y	235
MP Dumer	1.98	38.33	21.75y	5.9
MP MMT	0.48	33.83	3.69y	–

Next, we compared the expected runtime for a large SDP instance SDP(550, 275, 67) between MMT, multiparallel (MP) MMT and MP Dumer [15], which is an existing GPU-optimized ISD algorithm. Since this instance was unsolved, we compared the expected runtime TP^{-1} per server with a Tesla V100 GPU and an AMD Ryzen CPU calculated from the runtime T required for one loop in Algorithm 1 and the success probability P (or P') of each algorithm. The parameters used for MMT and MP Dumer were $p = 4, \ell_1 = 6, \ell_2 = 20$ and $p = 3, \ell = 19$, respectively. We used $p = 4, \ell_1 = 8$ and $\ell_2 = 18$ for the MP MMT, which are experimentally optimal values. Table 7 shows the result.

Finally, we performed several experiments on unresolved SDP instances in the Decoding Challenge [19]. We chose the following instances: SDP(510, 255, 62), SDP(530, 265, 65), SDP(540, 270, 66) and SDP(550, 275, 67). We used $p = 4, \ell_1 = 6, \ell_2 = 21$ for SDP(510, 255, 62) and $p = 4, \ell_1 = 8, \ell_2 = 18$ for the others. The expected runtimes per one Tesla V100 for each instance are 153.6, 219.8, 461.5 and 1350 days, respectively. We solved SDP(510, 255, 62) and SDP(530, 265, 65), taking 24.7 and 12.5 actual days, respectively, using four Tesla V100 servers. Moreover, we solved SDP(540, 270, 66), which took 79.44 days, using 22 Tesla V100 servers and solved SDP(550, 275, 67), which took 13.03 days, using four Tesla V100 servers. This result was posted as an official record on the Decoding Challenge website.

5.4 Comparison with Other ISD Implementations and SDP Instances

On the Decoding Challenge website, Meyer solved SDP(500, 250, 61) instance in 20.3 GPU-days using an ISD of Dumer's variant with Tesla V100 server(s). We achieved the expected runtime of 37.62 days for this instance using the multiparallel MMT with a single Tesla V100 server with optimal parameters $p = 8, \ell_1 = 7$ and $\ell_2 = 19$.

We also conducted an experiment on the McEliece instances in the Decoding Challenge website. Recently, SDP(1284, 1028, 24) in the McElice setting was solved by Esser, May and Zweydinger using their fast ISD implementation based on CPU parallelism [16]. They reported that the expected runtime to solve the instance with 4 AMD EPYC 7742 processors (512 threads) is 37.47 days and they solved the instance in 31.43 days. The expected runtime of the multiparallel MMT with 4 Tesla V100 servers is 158.22 days with optimal parameters $p = 8, \ell_1 = 10, \ell_2 = 26$, which is slower than their estimation. One reason for this is the high memory consumption of our algorithm. The memory required to solve SDP(530, 264, 65), which has a similar complexity to SDP(1284, 1028, 24), is only 395 MB, while the memory required for SDP(1284, 1028, 24) is 16.51 GB. If a smaller parameter $p = 4$ is chosen, the number of combinations for the base list L_{12} becomes $(k+\ell)/2$, which does not achieve a sufficient number of parallels for GPU. In such cases, simple instance parallelization would be more effective than intra-algorithm parallelization.

5.5 Analysis of the Number of Solutions

Finally, we performed an analysis on the distribution of the number of solutions for SDP instances. The motivation for this subsection is to confirm that luck in random seed selection does not play an important role in practice. The expected number of solutions for SDP(n, k, w) is given by $\binom{n}{w}/2^{n-k}$. We consider $k = n/2$ and $\binom{n}{w} > 2^{n-k}$, i.e., there exist multiple solutions. Especially, the weight parameter w is set to $w = \lceil 1.05 d_{\mathrm{GV}} \rceil$ as SDP instances in the Decoding Challenge, which is slightly higher than the Gilbert–Varshamov distance. The Gilbert–Varshamov distance is the smallest integer d satisfying $\sum_{j=0}^{d-1}\binom{n}{j} \geq 2^{n-k}$. By setting $w = \lceil 1.05 d_{\mathrm{GV}} \rceil$, there exist at least one solution with very high probability [19]. However, the actual number of solutions varies depending on the instance (random seeds for $\mathbf{H}$ and $\mathbf{s}$), we analyzed how the number of solutions is distributed to investigate the influence of the seed on the runtime of ISD.

As the result, we confirmed that the distribution of the number of solutions for SDP(n, k, w) fits the binomial distribution $B(N, p)$, where $N = \binom{n}{w}$ and $p = 1/2^{n-k}$. For instance, the distribution for SDP(550, 275, 67) is $B(2^{289.57}, 2^{-275})$. The variance $\sigma^2 = Np(1-p) = 27433.8$. The standard deviation $\sigma = 165.6$. Since the expected number of solutions is $Np = 27433.8$, the actual number of solutions may vary by up to 3%. When $p \ll 1$, the variance can be approximated by Np. Therefore, it was verified that luck by random seeds has less impact on the actual runtime of an ISD algorithm for large SDP instances.

6. Conclusion

In this paper, we analyzed the computational complexity of the modern ISD algorithms. We showed that the computational complexity of the MMT algorithm is lower than those of other ISDs when available memory is limited by using the Esser's syndrome decoding estimator. Furthermore, we proposed the multiparallel MMT algorithm, an optimized variant of MMT for multiparallel environments such as GPUs. In our experiment, we presented that the multiparallel MMT algorithm is faster than existing ISD implementations in several problem settings. In addition, we succeeded in solving four unresolved SDP instances on the Decoding Challenge website using the multiparallel MMT algorithm. A future work is to generalize the multiparallel MMT implementation in terms of parameter p so that it can be used as the multiparallel BJMM, which has a smaller WF than MMT for a larger amount of memory.

References

[1] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," IEEE Trans. Inf. Theory, vol.24, no.3, pp.384–386, 1978.
[2] I. Dumer, "On minimum distance decoding of linear codes," Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory, pp.50–52, 1991.
[3] A. May, A. Meurer, and E. Thomae, "Decoding random linear codes in $\tilde{O}(2^{0.054n})$," International Conference on the Theory and Application of Cryptology and Information Security, pp.107–124, 2011.
[4] A. Becker, A. Joux, A. May, and A. Meurer, "Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding," Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp.520–536, 2012.
[5] A. May and I. Ozerov, "On computing nearest neighbors with applications to decoding of binary linear codes," Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp.203–228, 2015.
[6] L. Both and A. May, "Decoding linear codes with high error rate and its impact for LPN security," International Conference on Post-Quantum Cryptography, pp.25–46, 2018.
[7] J. Coffey and R. Goodman, "The complexity of information set decoding," IEEE Trans. Inf. Theory, vol.36, no.5, pp.1031–1037, 1990.
[8] A. Barg, E. Krouk, and H. van Tilborg, "On the complexity of minimum distance decoding of long linear codes," IEEE Trans. Inf. Theory, vol.45, no.5, pp.1392–1405, 1999.
[9] E.A. Kruk, "Decoding complexity bound for linear block codes," Problemy Peredachi Informatsii, vol.25, no.3, pp.103–107, 1989.
[10] A. Esser and E. Bellini, "Syndrome decoding estimator," Public-Key Cryptography – PKC 2022, pp.112–141, 2022.
[11] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "A finite regime analysis of information set decoding algorithms," Algorithms, vol.12, no.10, p.209, 2019.
[12] Y. Hamdaoui and N. Sendrier, "A non asymptotic analysis of information set decoding," IACR Cryptol. ePrint Arch., vol.2013, p.162, 2013.
[13] C. Peters, "Information-set decoding for linear codes over $\mathbf{F}_q$," International Workshop on Post-Quantum Cryptography, pp.81–94, 2010.
[14] S. Heyse, R. Zimmermann, and C. Paar, "Attacking code-based cryptosystems with information set decoding using special-purpose hardware," PQCrypto 2014, pp.126–141, 2014.
[15] S. Narisada, K. Fukushima, and S. Kiyomoto, "Fast GPU implementation of Dumer's algorithm solving the syndrome decoding problem," IEEE ISPA 2021, pp.971–977, 2021.
[16] A. Esser, A. May, and F. Zweydinger, "McEliece needs a break — Solving McEliece-1284 and Quasi-Cyclic-2918 with modern ISD," Advances in Cryptology – EUROCRYPT 2022, pp.433–457, 2022.
[17] A. Esser, S. Ramos-Calderer, E. Bellini, J.I. Latorre, and M. Manzano, "An optimized quantum implementation of ISD on scalable quantum resources," arXiv preprint arXiv:2112.06157, 2021.
[18] S. Perriello, A. Barenghi, and G. Pelosi, "A complete quantum circuit to solve the information set decoding problem," 2021 IEEE International Conference on Quantum Computing and Engineering (QCE), pp.366–377, IEEE, 2021.
[19] N. Aragon, J. Lavauzelle, and M. Lequesne, "decodingchallenge.org," 2019. http://decodingchallenge.org
[20] E. Prange, "The use of information sets in decoding cyclic codes," IRE Trans. Inf. Theory, vol.8, no.5, pp.5–9, 1962.
[21] D. Wagner, "A generalized birthday problem," Annual International Cryptology Conference, pp.288–304, 2002.

Appendix: Reference Implementation in C++ CUDA

```
#include <array>
#include <bitset>
#include <cassert>
#include <fstream>
#include <iostream>
#include <random>
#include <thrust/copy.h>
#include <thrust/device_ptr.h>
#include <thrust/device_vector.h>
#include <thrust/scan.h>

using namespace std;

const int n = 550;
const int k = n / 2;
const int p = 8; // cannot be modified
const int N = ((n - k) + 63) / 64;
const int l1 = 18;
const int l2 = p;
const int l = l1 + l2;
const int mid = (n - k / 2 - l / 2);
const int size_L1 = ((k + l) / 2) * ((k + l) / 2 - 1) / 2;
const int size_L = pow(size_L1, 2) / pow(2, l2);
const int bucket_L1 = pow(2, l2);
const int bucket_L = pow(2, l1);
const int avg_L1 = size_L1 / bucket_L1;
const int num_L1 = avg_L1;
const int threads = 128;

#define CUDA_SAFE_FREE(x)   \
    if((x) != NULL) {       \
        cudaFree((x));      \
        (x) = NULL;         \
    }

int gaussian_elimination(array<bitset<n>, n - k> &h2, bitset<n - k> &s) {
    int rows = h2.size(), cols = h2[0].size();
    int rank = n - k - l;
    int r = 0;
    for(int c = 0; c < rank; c++) {
        int r2 = r;
        if(h2[r][cols - 1 - c] == 0) {
            for(; (r2 < rows) && (h2[r2][cols - 1 - c] == 0); ++r2) {
            }
            if(r2 >= rows) {
                return r;
            }
            swap(h2[r], h2[r2]);
            bool tmp = s[rows - 1 - r];
            s[rows - 1 - r] = s[rows - 1 - r2];
            s[rows - 1 - r2] = tmp;
        }
        for(int i = 0; i < rows; i++) {
            if(i == r)
                continue;
            if(h2[i][cols - 1 - c] == 1) {
                h2[i] = h2[i] ^ h2[r];
                s[rows - 1 - i] = s[rows - 1 - i] ^ s[rows - 1 - r];
            }
        }
        r = r + 1;
    }
    return r;
}

int Combination(int num, int r) {
    int c = 1;
    int b = 1;
    for(int i = 0; i < r; ++i) {
        c *= (num - i);
        b *= (i + 1);
    }

    return c / b;
}

void make_combination(int *comb, int start_index, int end_index, int step) {
    int *indexes = new int[step];
    int size = 0;
    int write_index = 0;
    while(size >= 0) {
        for(int i = start_index; i <= end_index; ++i) {
            indexes[size++] = i;
            if(size == step) {
                memcpy(&comb[write_index], indexes, sizeof(int) * step);
                write_index += step;
                break;
            }
        }
        if(--size < 0)
            break;
        start_index = indexes[size] + 1;
    }
    delete[] indexes;
}

bool MakeCombination(int nLeftCols, int nLeftComb, int *hLeftComb,
                     int *pdLeftComb, int nRightCols, int nRightComb,
                     int *hRightComb, int *pdRightComb) {
    try {
        make_combination(&hLeftComb[0], (n - k - l), mid - 1, p / 4);
        make_combination(&hRightComb[0], mid, n - 1, p / 4);

        cudaMemcpy(pdLeftComb, hLeftComb, sizeof(int) * nLeftComb * p / 4,
                   cudaMemcpyHostToDevice);
        cudaMemcpy(pdRightComb, hRightComb, sizeof(int) * nRightComb * p / 4,
                   cudaMemcpyHostToDevice);
    } catch(...) {
        return false;
    }
    return true;
}

bool InitializeDeviceMemory(u_int64_t **ppdH, int **ppdH1, int **ppdH2,
                            u_int64_t **ppdS, int nLeftComb, int **
                                   ppdLeftComb,
                            int nRightComb, int **ppdRightComb,
                            u_int64_t **ppdX, int **ppdL1, u_int64_t **ppdL,
                            u_int64_t **ppdR, int **ppdCounter, int **ppdE) {
    if(NULL != ppdH)
        cudaMalloc(ppdH, sizeof(u_int64_t) * N * (u_int64_t)n);
    if(NULL != ppdH1)
        cudaMalloc(ppdH1, sizeof(int) * (int)n);
    if(NULL != ppdH2)
        cudaMalloc(ppdH2, sizeof(int) * (int)n);
    if(NULL != ppdS)
        cudaMalloc(ppdS, sizeof(u_int64_t) * N);

    if(NULL != ppdLeftComb)
        cudaMalloc(ppdLeftComb, sizeof(int) * nLeftComb * p / 4);

    if(NULL != ppdRightComb)
        cudaMalloc(ppdRightComb, sizeof(int) * nRightComb * p / 4);
    if(NULL != ppdX)
        cudaMalloc(ppdX, sizeof(u_int64_t) * bucket_L * N);

    if(NULL != ppdL1)
        cudaMalloc(ppdL1, sizeof(int) * bucket_L1 * num_L1);

    if(NULL != ppdL)
        cudaMalloc(ppdL, sizeof(u_int64_t) * bucket_L);

    if(NULL != ppdR)
        cudaMalloc(ppdR, sizeof(u_int64_t) * bucket_L);

    if(NULL != ppdCounter)
        cudaMalloc(ppdCounter, sizeof(int) * bucket_L1);

    if(NULL != ppdE)
        cudaMalloc(ppdE, sizeof(int) * 8);

    return true;
}

void UninitializeDeviceMemory(u_int64_t **ppdH, int **ppdH1, int **ppdH2,
                              u_int64_t **ppdS, int **ppdLeftComb,
                              int **ppdRightComb, u_int64_t **ppdX, int **
                                     ppdL1,
                              u_int64_t **ppdL, u_int64_t **ppdR,
                              int **ppdCounter, int **ppdE){
    CUDA_SAFE_FREE((*ppdH)) CUDA_SAFE_FREE((*ppdH1)) CUDA_SAFE_FREE((*ppdH2))
        CUDA_SAFE_FREE((*ppdS)) CUDA_SAFE_FREE((*ppdLeftComb))
            CUDA_SAFE_FREE((*ppdRightComb)) CUDA_SAFE_FREE((*ppdX))
                CUDA_SAFE_FREE((*ppdL1)) CUDA_SAFE_FREE((*ppdL))
                    CUDA_SAFE_FREE((*ppdR)) CUDA_SAFE_FREE((*ppdCounter))
                        CUDA_SAFE_FREE((*ppdE))}

__global__ void GPU_compute_L(int *dCounter, int *pdH2, int *pdH1, int *pdL1,
                              u_int64_t *pdL, int nRightComb, int *pdLeftComb
                                     ,
                              int *pdRightComb, int num_L1) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < nRightComb) {
        int *pComb = pdRightComb + (p / 4 * idx);
        int h1 = *(pComb + 0);
        int h2 = *(pComb + 1);
        int x2 = pdH2[h1] ^ pdH2[h2];
        for(int i = 0; i < min(num_L1 - 1, dCounter[x2]); i++) {
            int idx2 = pdL1[x2 * num_L1 + i];
            int *lComb = pdLeftComb + (p / 4 * idx2);
            int hl1 = *(lComb + 0);
            int hl2 = *(lComb + 1);
            int x1 = pdH1[hl1] ^ pdH1[hl2] ^ pdH1[h1] ^ pdH1[h2];
            u_int64_t a = 0;
            a += ((u_int64_t)idx2 << 32);
            a += (idx);
            pdL[x1] = a;
        }
    }
}

__global__ void GPU_compute_R(int *dCounter, int *pdH2, int *pdH1, int *pdL1,
                              u_int64_t *pdR, int nRightComb, int *pdLeftComb
                                     ,
                              int *pdRightComb, int num_L1, int s2, int s1) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < nRightComb) {
        int *pComb = pdRightComb + (p / 4 * idx);
        int h1 = *(pComb + 0);
        int h2 = *(pComb + 1);
        int x2 = pdH2[h1] ^ pdH2[h2] ^ s2;
        for(int i = 0; i < min(num_L1 - 1, dCounter[x2]); i++) {
            int idx2 = pdL1[x2 * num_L1 + i];
            int *lComb = pdLeftComb + (p / 4 * idx2);
            int hl1 = *(lComb + 0);
            int hl2 = *(lComb + 1);
            int x1 = pdH1[hl1] ^ pdH1[hl2] ^ pdH1[h1] ^ pdH1[h2] ^ s1;
            u_int64_t a = 0;
            a += ((u_int64_t)idx2 << 32);
            a += (idx);
            pdR[x1] = a;
        }
    }
}
```

```
__global__ void GPU_match_LR(int bucket_L, int *pdH1, u_int64_t *pdH,
                             u_int64_t *pdL, u_int64_t *pdR, int *pdLeftComb,
                             int *pdRightComb, u_int64_t *s, int w, int *pdE,
                             u_int64_t *pdX) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < bucket_L) {
        if(pdL[idx] != 0 && pdR[idx] != 0) {
            int leftL = (pdL[idx] >> 32);
            int rightL = ((u_int32_t)~0 & pdL[idx]);
            int *l1i = pdLeftComb + (p / 4 * leftL);
            int *l2i = pdRightComb + (p / 4 * rightL);
            int e[8];
            e[0] = *(l1i + 0);
            e[1] = *(l1i + 1);
            e[2] = *(l2i + 0);
            e[3] = *(l2i + 1);
            int leftR = (pdR[idx] >> 32);
            int rightR = ((u_int32_t)~0 & pdR[idx]);
            int *r1i = pdLeftComb + (p / 4 * leftR);
            int *r2i = pdRightComb + (p / 4 * rightR);
            e[4] = *(r1i + 0);
            e[5] = *(r1i + 1);
            e[6] = *(r2i + 0);
            e[7] = *(r2i + 1);
            u_int64_t *x = &(pdX[N * idx]);
            for(int b = 0; b < N; b++) {
                x[b] = 0;
            }
            for(int i = 0; i < p; i++) {
                for(int b = 0; b < N; b++) {
                    x[b] = x[b] ^ pdH[N * e[i] + b];
                }
            }
            int diffs = 0;
            for(int b = 0; b < N; b++) {
                diffs += __popcll(x[b] ^ s[b]);
            }
            if(diffs <= w - p) {
                for(int i = 0; i < p; i++) {
                    pdE[i] = e[i];
                }
            }
        }
    }
}

array<int, n> MMT(array<array<u_int64_t, N>, n> h, array<u_int64_t, N> s, int
        w,
                  u_int64_t *pdH, int *pdH1, int *pdH2, u_int64_t *pdS,
                  int nLeftComb, int *hLeftComb, int *pdLeftComb,
                  int nRightComb, int *hRightComb, int *pdRightComb,
                  u_int64_t *pdX, int *pdL1, u_int64_t *pdL, u_int64_t *pdR,
                  int *dCounter, int *pdE) {
    auto error = cudaGetLastError();
    array<int, n> earray = {};
    u_int64_t *h_array = new u_int64_t[(u_int64_t)(n)*N];
    u_int64_t *pRefH = h_array;
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < N; ++j, pRefH++) {
            (*pRefH) = h[i][j];
        }
    }
    cudaMemcpy(pdH, h_array, sizeof(u_int64_t) * n * N,
           cudaMemcpyHostToDevice);

    int *h2 = new int[n];
    for(int i = 0; i < n; i++)
        h2[i] = 0;
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < l2; j++) {
            if(((h[i][N - 1] >> j) & 1) == 1)
                h2[i] += (1 << j);
        }
    }
    cudaMemcpy(pdH2, h2, sizeof(int) * n, cudaMemcpyHostToDevice);

    int *h1 = new int[n];
    for(int i = 0; i < n; i++)
        h1[i] = 0;
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < l1; j++) {
            if(((h[i][N - 1] >> (j + l2)) & 1) == 1)
                h1[i] += (1 << j);
        }
    }
    cudaMemcpy(pdH1, h1, sizeof(int) * n, cudaMemcpyHostToDevice);

    u_int64_t *s_array = new u_int64_t[N];
    for(int j = 0; j < N; ++j) {
        s_array[j] = s[j];
    }
    cudaMemcpy(pdS, s_array, sizeof(u_int64_t) * N, cudaMemcpyHostToDevice);

    int s2 = 0;
    for(int j = 0; j < l2; j++) {
        if(((s[N - 1] >> j) & 1) == 1)
            s2 += (1 << j);
    }
    int s1 = 0;
    for(int j = 0; j < l1; j++) {
        if(((s[N - 1] >> (j + l2)) & 1) == 1)
            s1 += (1 << j);
    }

    cudaMemset(pdL1, 0, sizeof(int) * bucket_L1 * num_L1);
    cudaMemset(dCounter, 0, sizeof(int) * bucket_L1);

    int *counterL1 = new int[bucket_L1];
    for(int i = 0; i < bucket_L1; i++)
        counterL1[i] = 0;
    int *L1 = new int[bucket_L1 * num_L1];
    for(int i = 0; i < bucket_L1 * num_L1; i++)
        L1[i] = 0;
    int x2 = 0;
    for(int i = 0; i < nLeftComb; i++) {
        int *pComb = hLeftComb + (p / 4 * i);
        int i1 = *(pComb + 0);
        int i2 = *(pComb + 1);
        x2 = h2[i1] ^ h2[i2];
        L1[x2 * num_L1 + min(num_L1 - 1, counterL1[x2])] = i;
        counterL1[x2] += 1;
    }
    cudaMemcpy(dCounter, counterL1, sizeof(int) * bucket_L1,
               cudaMemcpyHostToDevice);
    cudaMemcpy(pdL1, L1, sizeof(int) * bucket_L1 * num_L1,
               cudaMemcpyHostToDevice);

    cudaMemset(pdL, 0, sizeof(u_int64_t) * bucket_L);
    GPU_compute_L<<<(nRightComb + threads - 1) / threads, threads>>>(
        dCounter, pdH2, pdH1, pdL1, pdL, nRightComb, pdLeftComb, pdRightComb,
        num_L1);
    cudaDeviceSynchronize();

    cudaMemset(pdR, 0, sizeof(u_int64_t) * bucket_L);
    GPU_compute_R<<<(nRightComb + threads - 1) / threads, threads>>>(
        dCounter, pdH2, pdH1, pdL1, pdR, nRightComb, pdLeftComb, pdRightComb,
        num_L1, s2, s1);
    cudaDeviceSynchronize();

    int *e = new int[8];
    for(int i = 0; i < 8; i++)
        e[i] = -1;
    cudaMemset(pdE, -1, sizeof(int) * 8);
    cudaMemset(pdX, 0, sizeof(u_int64_t) * N * bucket_L);
    GPU_match_LR<<<(bucket_L + threads - 1) / threads, threads>>>(
        bucket_L, pdH1, pdH, pdL, pdR, pdLeftComb, pdRightComb, pdS, w, pdE,
        pdX);
    cudaDeviceSynchronize();
    cudaMemcpy(e, pdE, sizeof(int) * 8, cudaMemcpyDeviceToHost);

    if(e[0] != -1) {
        for(int i = 0; i < p; i++) {
            earray[e[i]] ^= 1;
        }
        array<uint64_t, N> diff = {0};
        for(int i = 0; i < p; i++) {
            if(e[i] != -1) {
                for(int b = 0; b < N; b++)
                    diff[b] = diff[b] ^ h[e[i]][b];
            }
        }
        for(int b = 0; b < N; b++)
            diff[b] = diff[b] ^ s[b];

        int diffs = 0;
        for(int b = 0; b < N; b++)
            diffs += __builtin_popcountll(diff[b]);

        uint64_t mask;
        int j = n - k - 1;
        for(int b = N - 1; b >= 0; b--) {
            for(int i = 0; i < 64; i++) {
                mask = (1ULL << i);
                mask = mask & diff[b];
                if(mask != 0ULL) {
                    earray[j] = 1;
                }
                j--;
                if(j < 0)
                    break;
            }
        }
    }
    delete[] h_array;
    delete[] h2;
    delete[] h1;
    delete[] s_array;
    delete[] counterL1;
    delete[] L1;
    delete[] e;
    return earray;
}

int main() {
    int w;
    bitset<n - k> s;
    bitset<n - k> s_mirror;
    bitset<n - k> x2;
    bitset<n> e;
    array<int, n> earray = {};
    array<bitset<n>, n - k> h;
    array<bitset<n>, n - k> h2;
    array<array<uint64_t, N>, n> hb;
    array<uint64_t, N> sb;
    u_int64_t *pdH = NULL;
    int *pdH1 = NULL;
    int *pdH2 = NULL;
    u_int64_t *pdS = NULL;
    int nLeftComb = 0;
    int *pdLeftComb = NULL;
    int nRightComb = 0;
    int *pdRightComb = NULL;
    int *pdL1 = NULL;
    int *dCounter = NULL;
    u_int64_t *pdL = NULL;
    u_int64_t *pdR = NULL;
    u_int64_t *pdX = NULL;
    int *pdE = NULL;
```

```
    string input_path = "./Challenges/SD/SD_" + to_string(n);
    ifstream in(input_path);
    cin.rdbuf(in.rdbuf());
    for(int i = 0; i < 9 + k; i++) {
        string input_string;
        int bit;
        getline(cin, input_string);
        if(i == 5) {
            w = stoi(input_string);
        } else if(i >= 7 && i < 7 + k) {
            for(int j = 0; j < n - k; j++) {
                bit = int(input_string[j]) - 48;
                h[j][(k - 1) - (i - 7)] = bit;
            }
        } else if(i == 8 + k) {
            s = bitset<n - k>(input_string);
            for(int j = 0; j < n - k; j++) {
                bit = int(input_string[j]) - 48;
                s_mirror[n - k - 1 - j] = bit;
            }
        }
    }
    for(int i = 0; i < n - k; i++) {
        h[i][n - 1 - i] = 1;
    }
    random_device rnd;
    mt19937 mt(rnd());
    uniform_int_distribution<> randn(0, n - 1);
    array<int, n> v;
    iota(v.begin(), v.end(), 0);
    int target, tmp;
    int nLeftCols = (mid - (n - k - 1));
    int nRightCols = (n - mid);
    nLeftComb = Combination(nLeftCols, 2);
    nRightComb = Combination(nRightCols, 2);
    int *hLeftComb = new int[(int)nLeftComb * p / 4];
    int *hRightComb = new int[(int)nRightComb * p / 4];

    InitializeDeviceMemory(&pdH, &pdH1, &pdH2, &pdS, nLeftComb, &pdLeftComb,
                           nRightComb, &pdRightComb, &pdX, &pdL1, &pdL, &pdR,
                           &dCounter, &pdE);

    MakeCombination(nLeftCols, nLeftComb, hLeftComb, pdLeftComb, nRightCols,
                    nRightComb, hRightComb, pdRightComb);

    while(true) {
        int rank;
        while(true) {
            for(int i = 0; i < n; i++) {
                target = randn(mt);
                tmp = v[target];
                v[target] = v[n - 1 - i];
                v[n - 1 - i] = tmp;
            }
            x2 = s_mirror;
            for(int i = 0; i < n - k; i++) {
                for(int j = 0; j < n; j++)
                    h2[i][n - 1 - j] = h[i][n - 1 - v[j]];
            }
            rank = gaussian_elimination(h2, x2);
            if(rank == n - k - 1)
                break;
        }
        for(int i = 0; i < n; i++) {
            int m = n - k - 1;
            for(int b = N - 1; b >= 0; b--) {
                hb[i][b] = 0ULL;
                for(int j = 0; j < 64; j++) {
                    if(h2[m][n - 1 - i]) {
                        hb[i][b] += (1ULL << j);
                    }
                    m--;
                    if(m < 0)
                        break;
                }
            }
        }
        int m = n - k - 1;
        for(int b = N - 1; b >= 0; b--) {
            sb[b] = 0ULL;
            for(int j = 0; j < 64; j++) {
                if(x2[n - k - 1 - m]) {
                    sb[b] += (1ULL << j);
                }
                m--;
                if(m < 0)
                    break;
            }
        }
        earray = MMT(hb, sb, w, pdH, pdH1, pdH2, pdS, nLeftComb, hLeftComb,
                     pdLeftComb, nRightComb, hRightComb, pdRightComb, pdX,
                            pdL1,
                     pdL, pdR, dCounter, pdE);

        for(int i = 0; i < n; i++) {
            if(earray[i] == 1)
                goto sol;
        }
    }
sol:
    int j = 0;
    for(int i : v) {
        e[n - 1 - i] = earray[j];
        j++;
    }
    bitset<n - k> He;
    array<bitset<n - k>, n> h_col;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n - k; j++)
            h_col[n - 1 - i][n - k - 1 - j] = h[j][i];
    for(int i = 0; i < n; i++) {
        if(e[n - 1 - i] == 1) {
            He = He ^ h_col[i];
        }
    }
    cout << " e (answer): " << e << endl;
    assert(He == s);
    delete[] hLeftComb;
    delete[] hRightComb;
    UninitializeDeviceMemory(&pdH, &pdH1, &pdH2, &pdS, &pdLeftComb,
                             &pdRightComb, &pdX, &pdL1, &pdL, &pdR, &dCounter
                                    ,
                             &pdE);
    return 0;
}
```

Shintaro Narisada received his B.E. in Electrical, Information and Physics Engineering and his M.E. in Information Sciences from Tohoku University, Japan, in 2016 and 2018, respectively. He joined KDDI in 2018 and has been engaged in the research on lattice-based cryptography. He is currently an associate research engineer at the Information Security Laboratory of KDDI Research, Inc.

Kazuhide Fukushima received his M.E. in Information Engineering from Kyushu University, Japan, in 2004. He joined KDDI and has been engaged in the research on post-quantum cryptography, cryptographic protocols, and identification technologies. He is currently a senior manager at the Information Security Laboratory of KDDI Research, Inc. He received his Doctorate in Engineering from Kyushu University in 2009. He received the IEICE Young Engineer Award in 2012. He served as the Editor of IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences from 2015 to 2017 and Director, General Affairs of IEICE Engineering Science Society from 2019 to 2021. He is a member of the Information Processing Society of Japan.

Shinsaku Kiyomoto received his B.E. in engineering sciences and his M.E. in Material Science from Tsukuba University, Japan, in 1998 and 2000, respectively. He joined KDD (now KDDI) and has been engaged in research on stream ciphers, cryptographic protocols, and mobile security. He is currently an executive director at KDDI Research, Inc. He was a visiting researcher of the Information Security Group, Royal Holloway University of London from 2008 to 2009. He received his doctorate in engineering from Kyushu University in 2006. He received the IEICE Young Engineer Award in 2004 and Distinguished Contributions Awards in 2011. He is a member of IEICE and JPS.