Faster Constant-Time Decoder for MDPC Codes and Applications to BIKE KEM

Thales B. Paiva and Routo Terada

University of Sao Paulo, Sao Paulo, Brazil, {tpaiva,rt}@ime.usp.br

Abstract. BIKE is a code-based key encapsulation mechanism (KEM) that was recently selected as an alternate candidate by the NIST's standardization process on post-quantum cryptography. This KEM is based on the Niederreiter scheme instantiated with QC-MDPC codes, and it uses the BGF decoder for key decapsulation. We discovered important limitations of BGF that we describe in detail, and then we propose a new decoding algorithm for QC-MDPC codes called PickyFix. Our decoder uses two auxiliary iterations that are significantly different from previous approaches and we show how they can be implemented efficiently. We analyze our decoder with respect to both its error correction capacity and its performance in practice. When compared to BGF, our constant-time implementation of PickyFix achieves speedups of 1.18, 1.29, and 1.47 for the security levels 128, 192 and 256, respectively.

Keywords: Post-quantum cryptography · BIKE · MDPC · LDPC · constant-time decoding

1 Introduction

BIKE [ABB+21] is a code-based key encapsulation mechanism (KEM) selected as an alternate candidate for the NIST post quantum standardization process. The scheme consists of a variant of the Niederreiter [Nie86] scheme using quasi-cyclic moderate-density parity-check (QC-MDPC) codes instead of Goppa codes. As such, BIKE can be seen as a refinement of Misoczki's et al. QC-MDPC McEliece [MTSB13].

The use of QC-MDCP [MTSB13] codes yields two advantages. The first one is that the public key is much smaller, since one needs only one row to represent a quasi-cyclic matrix in systematic form. The second is that matrix multiplication, and thus encoding, is much faster for quasi-cyclic matrices. However, QC-MDPC codes comes with an important disadvantage: their decoding algorithms have a non-zero probability of failure. This fact was exploited in the famous GJS [GJS16] key-recovery reaction attack, that provided the ground for side-channel attacks against QC-MDPC [RHHM17] and further attacks against other code-based encryption schemes [SSPB19, FHS+17].

To deal with this problem, BIKE's original proposal [ABB+17] used ephemeral keys. However, recent approaches on obtaining negligible decryption failure rate (DFR) [Til18, SV20a, Vas21], together with Hofheinz et al. [HHK17] CCA security conversions that accounts for decryption errors, motivated BIKE proponents to consider key-reuse. In particular, Sendrier and Vasseur [SV20a, Vas21] propose a framework that, under reasonable assumptions, allows them to find parameters where the DFR should be negligible using experiments and statistical analysis. This framework was used in BIKE's last revision [ABB+21], which uses the state-of-the-art BGF decoder [DGK20c, DGK19] with parameters that supposedly achieve negligible DFR.

While trying to improve BGF's performance, we noticed two limitations. The first one is that its performance cannot be improved by considering a lower number of iterations,

otherwise it breaks the main hypothesis for using Vasseur's extrapolation framework [Vas21]. The second is that some of its iterations can be made more efficient by merging them into one iteration. After analyzing BGF's strengths and weaknesses, we were able to derive a new and more efficient decoder.

Contribution. We propose a new decoding algorithm for QC-MDPC codes, called PickyFix. This decoder uses two auxiliary iterations that are significantly different from previous approaches: the FixFlip iteration, which flips a fixed number of bits, and the PickyFlip iteration, which uses different thresholds to flip ones and zeros. These iterations allow PickyFix to work with a lower number of iterations than BGF, which, together with our constant-time implementation, makes PickyFix achieve speedups of 1.18, 1.29, and 1.47 for the security levels 128, 192 and 256, respectively. The code and data are publicly available at https://github.com/thalespaiva/pickyfix.

Organization. We begin by quickly reminding some basic concepts from coding theory and QC-MDPC decoding in Section 2. BIKE and its security parameters are presented in Section 3. Then, in Section 4, we analyze BGF in detail to show its strengths and weaknesses. Our proposed decoder PickyFix is introduced in Section 5, and then we analyze its parameters and decoding performance in Section 6. In Section 7, we discuss how to implement PickyFix efficiently and in constant-time and then compare its performance with BGF. Finally, we conclude and discuss interesting future work in Section 8.

2 Background

A *binary* $[n, k]$*-linear code* is a k-dimensional linear subspace of $\mathbb{F}_2^n$, where $\mathbb{F}_2$ denotes the binary field. If $\mathcal{C}$ is a binary $[n, k]$-linear code spanned by the rows of a matrix $\mathbf{G}$ of $\mathbb{F}_2^{k\times n}$, we say that $\mathbf{G}$ is a *generator matrix* of $\mathcal{C}$. Similarly, if $\mathcal{C}$ is the kernel of a matrix $\mathbf{H}$ of $\mathbb{F}_2^{r\times n}$, we say that $\mathbf{H}$ is a *parity-check matrix* of $\mathcal{C}$. The Hamming *weight* of a vector $\mathbf{v}$, denoted by $|\mathbf{v}|$, is the number of its non-zero entries. The *syndrome* $\mathbf{z}$ of a vector $\mathbf{e}$ with respect to a parity check matrix $\mathbf{H}$ is the vector $\mathbf{z} = \mathbf{e}\mathbf{H}^\top$. If the vector $\mathbf{e}$ is sufficiently sparse and the linear code defined by $\mathbf{H}$ is sufficiently good, it may be possible to recover $\mathbf{e}$ from the syndrome $\mathbf{z}$ by using efficient decoding algorithms. The support of a binary vector $\mathbf{v}$, denoted as $\mathrm{supp}(\mathbf{v})$, is the set $\mathrm{supp}(\mathbf{v}) = \{i : \mathbf{v}_i = 1\}$.

A moderate-density parity-check (MDPC) code [MTSB13] is a linear code that admits a moderately sparse parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{r\times n}$. The weight of each column of $\mathbf{H}$ is set to be all equal to a fixed value d, and require that $d = O(\sqrt{n})$. For applications in cryptography, it is particularly useful to consider quasi-cyclic MDPC (QC-MDPC) codes, because they allow for smaller keys and more efficient operations. BIKE [ABB+21] is defined over QC-MDPC codes with two circulant blocks, which are MDPC codes that admit a sparse parity check matrix of the form $\mathbf{H} = [\mathbf{H}_0|\mathbf{H}_1]$, where each $r \times r$ binary matrix $\mathbf{H}_0$ and $\mathbf{H}_1$ is circulant.

MDPC codes admit very efficient decoders, which are called bit-flipping decoders [Gal62]. All variants of bit-flipping decoders work based on the following observations. Let $\mathbf{e}$ be a sparse vector whose syndrome with respect to the sparse matrix $\mathbf{H}$ is $\mathbf{z} = \mathbf{e}\mathbf{H}^\top$. Suppose we do not know $\mathbf{e}$ but want to recover it from $\mathbf{z}$ using our knowledge from $\mathbf{H}$. We know that $\mathbf{z} = \sum_{i\in\mathrm{supp}(\mathbf{e})} \mathbf{H}_i^\top$, where $\mathbf{H}_i^\top$ denotes the transpose of the i-th column of $\mathbf{H}$.

Now, since $\mathbf{e}$ and each column $\mathbf{H}_i^\top$ are sparse, we can estimate the likelihood that $\mathbf{e}_i = 1$ by checking how closely $\mathbf{z}$ matches with column i of $\mathbf{H}$: the more they are similar, the higher is the probability that $\mathbf{e}_i = 1$. The similarity measure for each column i is what is known as the unsatisfied parity-check (UPC) counter, denoted as $\mathtt{upc}_i$, and it is equal to the size of the intersection of $\mathrm{supp}(\mathbf{z})$ and $\mathrm{supp}(\mathbf{H}_i^\top)$. The name UPC comes from the

Algorithm 1 General bit-flipping decoding algorithm.

procedure GeneralBitFlipping($\mathbf{z}, \mathbf{H}$)
  Start with the partial error vector $\hat{\mathbf{e}} \leftarrow \mathbf{0} \in \mathbb{F}_2^n$
  Initialize the number of iterations $\mathtt{it} \leftarrow 0$
  Let the partial syndrome $\mathbf{s} \leftarrow \mathbf{z} + \hat{\mathbf{e}}\mathbf{H} = \mathbf{z}$
  while $\mathbf{s} \neq \mathbf{0}$ and $\mathtt{it} <$ maximum number of iterations **do**
    Compute the UPC counters $\mathtt{upc}_i$ with respect to $\mathbf{s}$ and $\mathbf{H}$, for $i = 1$ to n
    For each $i = 1$ to n, flip bit $\hat{\mathbf{e}}_i$ if $\mathtt{upc}_i$ is above a certain threshold
    Update the partial syndrome $\mathbf{s} \leftarrow \mathbf{z} + \hat{\mathbf{e}}\mathbf{H}$
    $\mathtt{it} \leftarrow \mathtt{it} + 1$
  if $\mathbf{s} = \mathbf{0}$ **then**
    return $\hat{\mathbf{e}}$
  else
    return $\perp$, indicating that the maximum number of iterations was reached

fact that the set $\mathrm{supp}(\mathbf{z})$ is sometimes called the set of unsatisfied equations, and therefore $\mathtt{upc}_i$ counts the number of unsatisfied equations that are caught by $\mathbf{H}_i^\top$.

Algorithm 1 shows the steps that a general bit-flipping algorithm performs when trying to obtain $\mathbf{e}$ from $\mathbf{z}$ and $\mathbf{H}$. The algorithm stops when it finds a vector $\hat{\mathbf{e}}$ with the same syndrome as $\mathbf{e}$, or if the number of iterations exceeds some limit. Notice that the partial syndrome defines the objective syndrome in each iteration, and in the ideal case, vector $\hat{\mathbf{e}}$ gets closer and closer to $\mathbf{e}$ after each iteration. Although most bit-flipping algorithms used in cryptography [Gal62, MTSB13, DGK19, SV19] can be framed in the general description above, they can vary significantly with respect to how the threshold for flipping bits is selected in each iteration.

3 BIKE

The purpose of a key encapsulation mechanism is to use public-key encryption algorithms to securely exchange a key between two parties. These parties can then use secret-key algorithms, which are much more efficient, to exchange large messages.

For a clearer presentation, we describe BIKE algorithms without the implicit-rejection Fujisaki-Okamoto transformation [HHK17], usually denoted by $\mathrm{FO}^{\not\perp}$. However, notice that when discussing the experimental performance of our algorithm in Section 7.3, we consider the full decapsulation with the $\mathrm{FO}^{\not\perp}$ transformation applied.

3.1 Parameters and Algorithms

Setup. On input 1^λ, where λ is the security level, the setup algorithm returns parameters r, w and t taken from the parameter Table 1. Parameters r and w will define the family of QC-MDPC codes to be used while t controls the weight of the error used for encryption, as will be detailed in the following sections. The table also provides the estimated decryption failure rates (DFR) for each parameters set according to Vasseur's framework [Vas21].

Table 1: BIKE parameters for each security level.

Parameter set	Security level λ	r	w	$d = w/2$	t	Decoder	DFR estimate
BIKE Level 1	128	12,323	142	71	134	BGF	2^{-128}
BIKE Level 3	192	24,659	206	103	199	BGF	2^{-192}
BIKE Level 5	256	40,973	274	137	264	BGF	2^{-256}

Key Generation. Let $\mathbf{h}_0$ and $\mathbf{h}_1$ be two vectors of r bits of odd weight $d = w/2$. Build the circulant matrices $\mathbf{H}_0$ and $\mathbf{H}_1$ by taking $\mathbf{h}_0$ and $\mathbf{h}_1$ as their first rows, correspondingly. If $\mathbf{H}_1$ is not invertible, restart the process by selecting another $\mathbf{h}_1$. Then the secret key is the sparse matrix $\mathbf{H} = [\,\mathbf{H}_0 \,|\, \mathbf{H}_1\,] \in \mathbb{F}_2^{r\times 2r}$ and the public key is the dense circulant matrix $\mathbf{H}_{\text{Pub}} = \mathbf{H}_1\mathbf{H}_0^{-1}$.

Notice that matrices $\mathbf{H}$ and $[\,\mathbf{I} \,|\, \mathbf{H}_{\text{Pub}}\,]$ are both parity checks of the same quasi-cyclic linear code. However, the sparsity of the first one allows for efficient syndrome decoding using bit-flipping algorithms.

Encapsulation. Select two random binary vectors $\mathbf{e}_0$ and $\mathbf{e}_1$ such that $|\mathbf{e}_0| + |\mathbf{e}_1| = t$. Then the key to be shared is $k_{\text{Shared}} = \mathcal{H}([\mathbf{e}_0|\mathbf{e}_1])$, for some cryptographic hash function $\mathcal{H}$. To encapsulate the key k_{Shared}, compute the ciphertext $\mathbf{c} = \mathbf{e}_0 + \mathbf{e}_1\mathbf{H}_{\text{Pub}}^\top \in \mathbb{F}_2^r$. Notice that ciphertext $\mathbf{c}$ then corresponds to the syndrome of the low weight vector $[\mathbf{e}_0|\mathbf{e}_1]$ with respect to the public parity-check matrix $[\,\mathbf{I} \,|\, \mathbf{H}_{\text{Pub}}\,]$.

Decapsulation. Given the ciphertext $\mathbf{c}$, the receiver, who knows the sparse parity-check matrix $\mathbf{H}$, first compute the secret syndrome $\mathbf{z} = \mathbf{c}\mathbf{H}_0^\top$. Notice that

$$\mathbf{z} = \mathbf{c}\mathbf{H}_0^\top = \left(\mathbf{e}_0 + \mathbf{e}_1\mathbf{H}_{\text{Pub}}^\top\right)\mathbf{H}_0^\top = \mathbf{e}_0\mathbf{H}_0^\top + \mathbf{e}_1\mathbf{H}_{\text{Pub}}^\top\mathbf{H}_0^\top = \mathbf{e}_0\mathbf{H}_0^\top + \mathbf{e}_1\mathbf{H}_1^\top.$$

Therefore, as mentioned by the end of Section 2, the receiver can use some QC-MDPC bit-flipping decoding algorithm, together with their knowledge of the secret matrix $\mathbf{H}$ to recover the sparse vector $[\mathbf{e}_0|\mathbf{e}_1]$ and compute the shared key $k_{\text{Shared}} = \mathcal{H}([\mathbf{e}_0|\mathbf{e}_1])$.

In the last revision of BIKE [ABB+21], the authors recommend the BGF decoding algorithm [DGK20c], which is the state-of-the art QC-MDPC decoder. Before introducing BGF, let us first discuss the security of BIKE and, in particular, why good decoders are very important to ensure BIKE's security.

3.2 Security and Negligible Decryption Failure Rate

The security of the scheme is based on three hypotheses. The first two are standard conjectures for quasi-cyclic codes, namely the hardness of the syndrome decoding problem and the hardness of finding codewords of a fixed low weight. This ensures that one can neither recover the secret sparse matrices $\mathbf{H}_0$ and $\mathbf{H}_1$ from $\mathbf{H}$, nor the secret message $[\mathbf{e}_0|\mathbf{e}_1]$ from ciphertext $\mathbf{c}$. The third hypothesis is that the decryption failure rate (DFR) is negligible with respect to the security parameter. Although we cannot prove the third hypothesis, Vasseur [Vas21] proposed a framework that, under weaker hypothesis, allows one to get confident that some decoders achieve negligible DFR for selected parameter sets.

It is shown [TS16, Sen11] that parameters t and w are the most important when determining the security level, since they control the weight of the sparse vectors. Intuitively, if w or t are too small, it is easy to find $\mathbf{h}_0$ or the partial encryption error $\mathbf{e}_0$ by enumerating low weight vectors. But they may not be so large, with respect to r, otherwise the probability of failing to decrypt a ciphertext would be too high. Therefore, to define parameters (t, w, r), one typically fixes (t, w) sufficiently large to achieve high security levels, and then define r such that the decryption failure rate is low enough for the desired application.

In 2016, Guo et al. [GJS16] showed that decryption failures could lead to a full key recovery attack against schemes based on QC-MDPC codes. To deal with the potential vulnerability faced by schemes within which decryption failures occur, Hofheinz et al. [HHK17] refined the Fujisaki-Okamoto [FO99] transformation showing that a scheme whose decryption failure rate is below $2^{-\lambda}$ can be transformed into a CCA secure one.

Unlike for algebraic codes, such as Goppa or Reed-Solomon codes, whose decoders are guaranteed to decode all errors in vectors up to a given weight, we cannot yet give strong

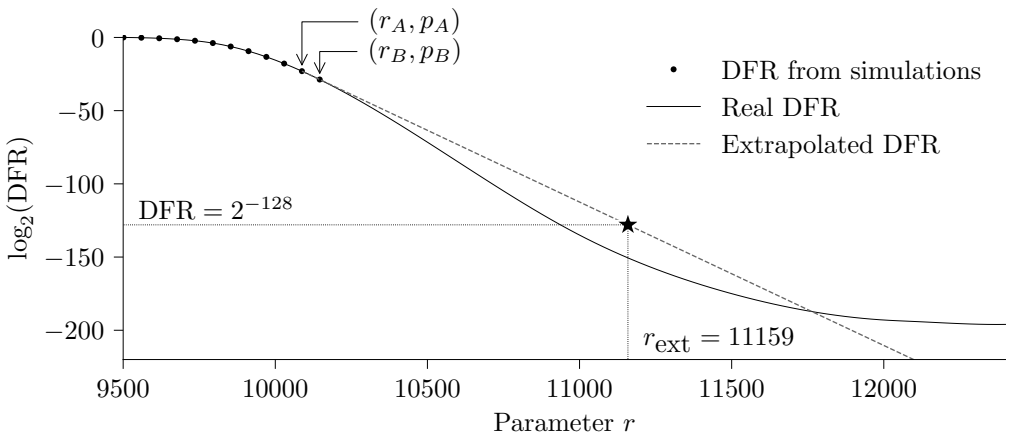

Figure 1: Illustration of Vasseur's [Vas21] DFR extrapolation framework considering 128 bits of security and a hypothetical decoder.

mathematical guarantees on the error correction capability of decoders for QC-MDPC codes. Recently, Sendrier and Vasseur [SV20a, Vas21] proposed a method that, under reasonable hypotheses, allows one to use simulations and simple statistical analysis to find parameters (r, t, w) such that a QC-MDPC decoder fails with negligible probability with respect to some security parameters λ.

Let t and w be fixed positive integers and let us consider a hypothetical QC-MDPC decoder $\mathcal{D}$. Let $\text{DFR}_{\mathcal{D}}(r)$ denote the decryption failure rate of $\mathcal{D}$ when decrypting a ciphertext generated at random with respect to a random QC-MDPC key with parameters (r, t, w). The main observation by Sendrier and Vasseur [SV20a] is that the curve $\log_2(\text{DFR}_{\mathcal{D}}(r))$ is typically concave for practical QC-MDPC decoders and for all values of r such that $\text{DFR}_{\mathcal{D}}(r)$ is high enough so that failures can be observed in simulations. Vasseur's [Vas21] model then makes the following assumption: for a given decoder $\mathcal{D}$ and security level λ, the curve $\log_2(\text{DFR}_{\mathcal{D}}(r))$ is concave in the region where $\text{DFR}_{\mathcal{D}}(r) \geq 2^{-\lambda}$. This assumption is somewhat consistent with Tillich's [Til18] asymptotic theoretical model for MDPC codes, which shows that the dominating term in $\log_2(\text{DFR}_{\mathcal{D}}(r))$ decreases linearly with r.

Figure 1 illustrates how Vasseur's [Vas21] model can be used to estimate the block parameter r that allows for negligible failure rate with respect to the security parameter $\lambda = 128$. First, one performs DFR simulations for increasing values of r until it cannot see any decoding failure. Then, they take the last two points (r_A, p_A) and (r_B, p_B) in the $\log_2$ DFR plot such that a number of failures were observed and compute the line passing through them. According to the extrapolation hypothesis, the decoder fails with negligible probability for $r = r_{\text{ext}}$, the point where the line intercepts $\text{DFR} = 2^{-\lambda}$. Finally, choose parameter r to be the least prime $r \geq r_{\text{ext}}$ such that 2 is primitive modulo r. This avoids both squaring attacks [LJS+16] and other potential attacks based on the factorization of the cyclic polynomial ring[1] $\mathbb{F}_2[X]/(X^r - 1)$.

Since there is always some error in the DFR estimates, Vasseur [Vas21] uses confidence intervals for the observed DFR and compute a conservative extrapolation for r as follows. Let p_A and p_B be the DFRs for r_A and r_B, respectively, where $r_A < r_B$. Consider p_A^- and p_B^+ to be the lower and upper limit for p_A and p_B according to Binomial confidence intervals for p_A and p_B. Then a conservative extrapolation for r_{ext} is obtained by considering the

[1]The cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$ is isomorphic to the ring of circulant matrices used in BIKE.

line passing through (r_A, p_A^-) and (r_B, p_B^+). Vasseur [Vas21] uses the Clopper-Pearson confidence interval together with posterior probabilities to obtain a narrower interval, with confidence level $\alpha = 0.01$. In this work we use the same α with the Clopper-Pearson interval, but we do not use the posterior probabilities. Even though this tends to give slightly more conservative estimates, it is easier to compute.

3.3 BGF: State-of-the-art QC-MDPC Decoder

BGF [DGK20c], which stands for Black-Gray-Flip, is one of the most efficient known decoders for QC-MDPC codes. This decoder is an improvement of the Black-Gray decoder first proposed by Sendrier and Misoczki in a previous version[2] of CAKE [BGG+17], a predecessor of BIKE.

As a decoding algorithm, BGF's goal is to, given a syndrome ciphertext $\mathbf{c} = \mathbf{e}_0 + \mathbf{e}_1\mathbf{H}_{\text{Pub}}^\top$, recover the sparse error vector $\mathbf{e} = [\mathbf{e}_0|\mathbf{e}_1]$ using the secret sparse matrix $\mathbf{H}$. The algorithm first computes the secret syndrome $\mathbf{z} = \mathbf{c}\mathbf{H}_0^\top$, then starts with $\mathbf{e} \leftarrow \mathbf{0}$ and performs a sequence of N_{Iter} iterations, each of which updates its knowledge on $\mathbf{e}$ until either $\mathbf{z} = \mathbf{e}\mathbf{H}^\top$ or the number of iterations exceeds a certain limit N_{Iter} and a decoding failure occurs. Before introducing BGF, let us first define its auxiliary procedures.

BGF Auxiliary Algorithms. BGF uses two bit-flipping auxiliary procedures: BitFlipIter and BitFlipMaskedIter, which are formally described in Algorithm 2. These procedures are very similar to other iterative decoders, such as the original Gallager's bit-flipping algorithm [Gal62].

Both algorithms flip bits of the partial error vector $\mathbf{e}$ when their corresponding UPC counters are above some threshold, τ_0 for BitFlipIter and τ_1 for BitFlipMaskedIter. However they differ in some important points. First, BitFlipIter not only flips the bits, but it also marks the bits in either black or gray, using bit-masks `BlackMask` and `GrayMask`. Black bits are the ones that are flipped with a somewhat high confidence ($\texttt{upc}_j \geq \tau_0$), while gray bits are the ones that were almost selected for flipping ($\tau_0 > \texttt{upc}_j \geq \tau_0 - \delta$), but did not make it because of a minor difference δ. On the other hand, BitFlipMaskedIter is a simple bit flip iteration based on the UPC value, but it only flips bits that are marked 1 in a given mask `Mask`.

The BGF algorithm. BGF is defined as Algorithm 3. Intuitively, the first call to BitFlipIter flips the bits for which it has a high confidence that they are wrong, by using a selective threshold function Thresh. Then it comes the two regret steps: first the black and then the gray. In the black regret, all the 1 bits added in the previous step that have an UPC strictly greater[3] than $(d+1)/2$ will be flipped back to 0. The gray regret step is analogous, but now over the bits marked in `GrayMask`, which are called gray bits. These consist of 0 bits that were not flipped in the first step because their UPC were smaller than, but somewhat close to, the selected threshold.

After the first and most costly iteration ensured a good start, hopefully with only a small number of errors left to be corrected, BGF continues with $N_{\text{Iter}} - 1$ iterations of BitFlipIter that will try to correct the remaining errors. Notice that the masks are not needed after this point, and thus, are ignored.

BGF Parameters. Table 2 shows the parameters δ, N_{Iter} and threshold function Thresh proposed for the different security levels together with their performance under our platform[4]. We considered the constant-time implementation provided in BIKE Additional

[2] Unfortunately, there appears to be no reference to the version in which the Black-Gray decoder appeared.

[3] Notice that this is done by choosing $\tau_1 = (d+1)/2 + 1$, since the flipping condition is $\texttt{upc}_j \geq \tau_1$.

[4] Intel® Xeon™ Gold 5118 CPU at 2.30GHz.

Algorithm 2 Auxiliary iterations used by BGF.

procedure BitFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0$)
  $\texttt{BlackMask} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$
  $\texttt{GrayMask} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$
  for $j = 1$ to $2r$ **do**
    $\texttt{upc}_j \leftarrow \left|\text{supp}\left(\mathbf{s}\right) \cap \text{supp}\left(\mathbf{H}_j^\top\right)\right|$
    if $\texttt{upc}_j \geq \tau_0$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$ ▷ Flips coordinate j of $\mathbf{e}$
      $\texttt{BlackMask}_j \leftarrow 1$
    else if $\texttt{upc}_j \geq \tau_0 - \delta$ **then**
      $\texttt{GrayMask}_j \leftarrow 1$
  $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
  return $\mathbf{e}$, $\mathbf{s}$, BlackMask, GrayMask

procedure BitFlipMaskedIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{Mask}, \tau_1$)
  for $j = 1$ to $2r$ **do**
    $\texttt{upc}_j \leftarrow \left|\text{supp}\left(\mathbf{s}\right) \cap \text{supp}\left(\mathbf{H}_j^\top\right)\right|$
    if $\texttt{Mask}_j = 1$ and $\texttt{upc}_j \geq \tau_1$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$ ▷ Flips coordinate j of $\mathbf{e}$
  $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
  return $\mathbf{e}$, $\mathbf{s}$

Implementation [DGK20a] with minor changes to account for the updated threshold function in BIKE's last revision [ABB+21].

Notice how δ and N_{Iter} are the same in all security levels. The threshold function is an increasing linear function on the syndrome weight truncated above the minimum value $(d+1)/2$. Since the threshold function is used to determine when to flip a bit, this means that when the weight of the syndrome $\mathbf{s}$ is large, fewer bits will be flipped.

Table 2: BGF parameters and their corresponding performance when considering the portable and AVX512 implementations.

Security level λ	δ	N_{Iter}	Thresh($\mathbf{s}$)	Cycles Portable	Cycles AVX512
128	3	5	$\max\left(36, \lfloor 0.00697220\,\lvert\mathbf{s}\rvert + 13.5300\rfloor\right)$	10,955,732	1,323,322
192	3	5	$\max\left(52, \lfloor 0.00526500\,\lvert\mathbf{s}\rvert + 15.2588\rfloor\right)$	32,982,825	4,130,087
256	3	5	$\max\left(69, \lfloor 0.00402312\,\lvert\mathbf{s}\rvert + 17.8785\rfloor\right)$	94,902,236	11,497,288

4 Critical Analysis of BGF

In this section, we dive a little deeper into the BGF decoding algorithm. This allows us to better understand why BGF is effective, but, more importantly, it will show some of BGF's weaknesses and lay the ground over which a better decoder can be designed. It is well-known to be difficult to provide a theoretical analysis for QC-MDPC iterative decoders, because of the inherent dependency caused by the circulant matrices involved. Therefore, our analysis is based on observations of BGF's behavior in practice.

4.1 BGF's First Iteration: The Black-Gray Step

Let us first discuss BGF's first iteration and its importance for the extrapolation framework. As described in the previous section, in the first iteration, BGF performs a sequence of 3

Algorithm 3 The BGF decoding algorithm.

procedure BGF($\mathbf{H} = [\mathbf{H}_0|\mathbf{H}_1], \mathbf{z} = \mathbf{c}\mathbf{H}_0^\top$)
  $\mathbf{e} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$ ▷ Initializes the partial error vector
  $\mathbf{s} \leftarrow \mathbf{z}$ ▷ Initializes the partial syndrome
  for $i = 1$ to N_{Iter} **do**
    ▷ Every time $\mathbf{e}$ and $\mathbf{s}$ are updated, it holds that $\mathbf{s} = \mathbf{z} + \mathbf{e}\mathbf{H}^\top$
    if $i = 1$ **then**
      $\mathbf{e}, \mathbf{s}, \texttt{BlackMask}, \texttt{GrayMask} \leftarrow$ BitFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0 =$ Thresh($\mathbf{s}$))
      $\mathbf{e}, \mathbf{s} \leftarrow$ BitFlipMaskedIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{BlackMask}, \tau_1 = (d+1)/2+1$)
      $\mathbf{e}, \mathbf{s} \leftarrow$ BitFlipMaskedIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \texttt{GrayMask}, \tau_1 = (d+1)/2+1$)
    else
      $\mathbf{e}, \mathbf{s} \leftarrow$ BitFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_0 =$ Thresh($\mathbf{s}$)) ▷ Ignores the black-gray masks
  if $\mathbf{e}\mathbf{H}^\top = \mathbf{z}$ **then** ▷ This condition is equivalent to $\mathbf{s} = \mathbf{0}$
    return $\mathbf{e}$
  else
    return $\perp$ ▷ Decoding failure

bit-flipping calls: one BitFlipIter followed by two BitFlipMaskedIter.

We know that BitFlipIter flips all bits whose UPC counters are above a certain threshold. This makes it very sensible to the threshold selected, as illustrated in Figure 2. Consider the difference if, by chance, the threshold $\tau_0 = 76$ was selected, then the number of errors made after calling BitFlipIter, that is, correct bits that would be incorrectly flipped, would be twice the number if $\tau_0 = 77$ were selected.

This problem is particularly important under the extrapolation framework, where the algorithm needs not only to perform well, but also to improve its performance at a very fast rate for small, but increasing, values of r. Therefore, BGF uses a very conservative threshold in the first iteration BitFlipIter. Additionally, the black and gray regretting phases, corresponding to the two calls of BitFlipMaskedIter, also work by flipping a controlled number of bits: only those bits in the black or gray masks whose UPC is above $(d+1)/2$. This makes the whole first iteration very conservative.

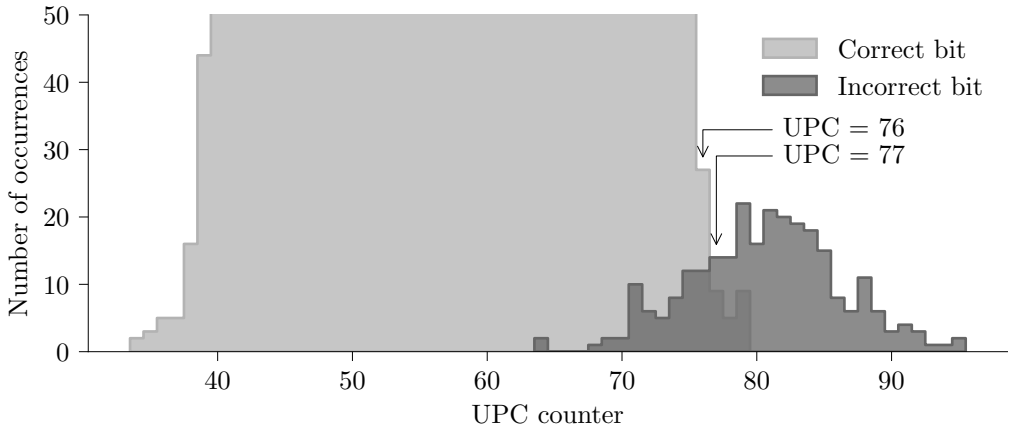

Figure 2: Histogram of the UPC counters for each of the $2r$ bits in the partial error vector $\mathbf{e} = \mathbf{0}$, in the beginning of the first iteration, separated by the cases when the bit is right or wrong. The values correspond to a real observation corresponding to the BIKE Level 5 security parameters.

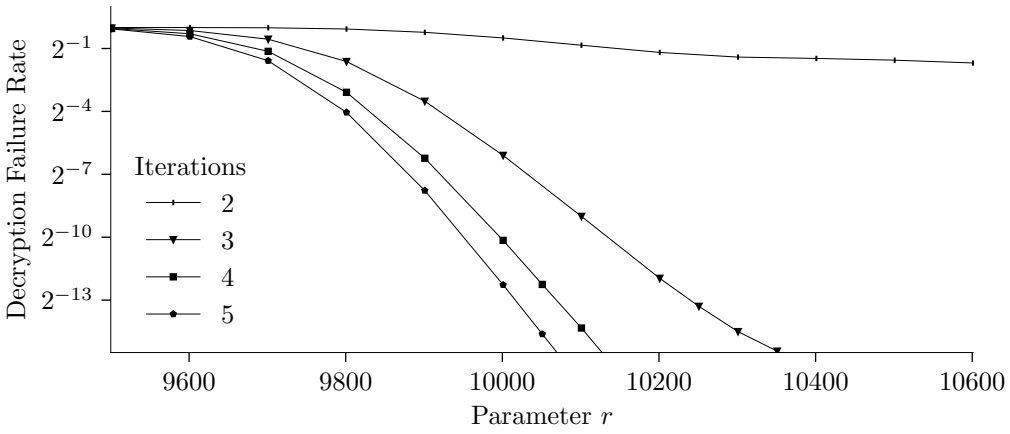

Figure 3: The impact of the number of BGF iterations on the DFR, considering parameter set BIKE Level 1 ($t = 134$, $w = 142$).

Even though a conservative first iteration is important to ensure a fast DFR decay when r increases, it may result in useless iterations when r is close to the value when negligible DFR is reached. In particular, for the case presented in Figure 2, where $r = 40{,}973$, Thresh returned $\tau_0 = 86$. This would result in no error being made after BitFlipIter, but at the cost of flipping only a small number of bits, compared to the case where $\tau_0 = 80$, for example.

This suggests that removing the black regret step may be a good starting point for optimization. For example, we could merge both black and gray regret steps into one iteration in such a way that the black regret is critical for small r, but when r gets larger, the gray regret steps gets more important than the black one. This is the key idea behind our PickyFlip iteration that we introduce in Section 5.

4.2 The Number of Iterations and the Threshold Function

One straightforward method to improve the decapsulation performance would be to decrease the number of BGF iterations, at the cost of increasing the key sizes. Intuitively, one may think that there is a direct trade-off between the number of iterations and the block length parameter r: the DFR may not decay as fast when using a lower number of iterations, but one might be lucky to obtain a reasonable value of r after the extrapolation. However, as discussed in the previous section, since the thresholds are so conservative, if the number of iterations is too small, the decoder may not be able to fully correct the errors even for large values of r.

Figure 3 shows how the number of iterations affects the decay of the DFR as a function of r. Notice how 2 iterations are not enough to allow for a complete decoding of errors of weight $t = 134$. Furthermore, the curve for 3 iterations does not appear to be concave, therefore it is not safe to use the extrapolation framework for this value. This odd behavior of the DFR curves for 2 and 3 iterations is caused by the following problem. On the one hand, increasing r should make it easier to correct more errors, since there is more redundancy, but, for large values of r, the threshold τ_0 used in the first iteration is so high that only very few errors are corrected in the first iteration.

Let us analyze the thresholds τ_0 in more detail. The average values of the thresholds τ_0 used in each of BGF's iteration are shown in Figure 4, considering 10,000 decapsulations under BIKE Level 1 parameter set. We make three observations. First notice how the first

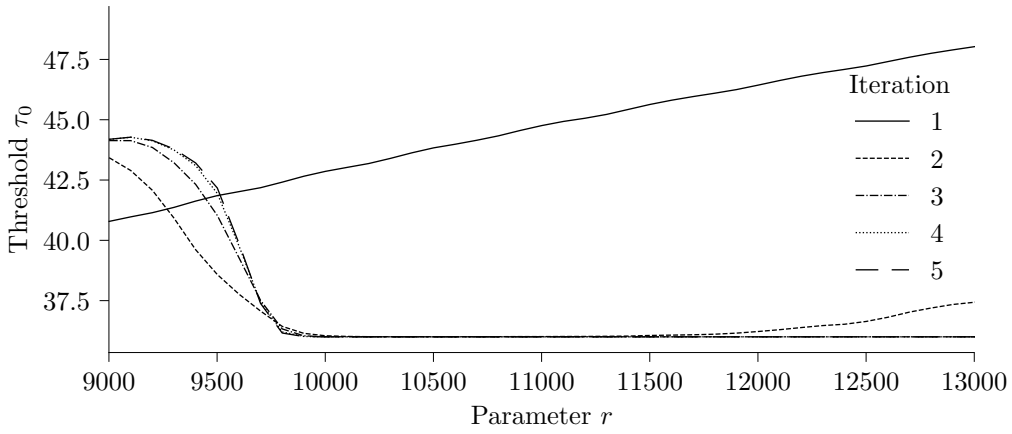

Figure 4: Average values of threshold τ_0 in each of the 5 iterations of BGF, considering parameter set BIKE Level 1 ($t = 134$, $w = 142$).

threshold increases as r increases. This is a consequence of the linear dependency of τ_0 on the syndrome weight $|\mathbf{s}|$, which turns out to increase with r. The second observation is that the thresholds used in iterations 2 to 5 appear to converge to the floor $(d+1)/2 = 36$. This happens because the first iteration, in general, is able to flip a sufficiently large number of errors, and leave only fine adjustments for the next iterations to deal with. The third is that τ_0, for the second iteration, starts increasing after $r = 11{,}500$. This is caused by the threshold in the first iteration being too high, which leaves a lot of errors to be corrected by the second iteration.

4.3 Impact of the Threshold on the Concavity Assumption

Back to the DFR curves, the non-concave behavior of the curves for 2 and 3 iterations raises a potentially deep problem with the BGF threshold: why should we expect the curves for 4 and 5 iterations to be concave as well? It is possible that we just cannot see an inflection point because it is located at a DFR smaller than what we can simulate.

To evaluate the concavity of the DFR curves for 5 iterations, we propose the following experiment. Consider BIKE Level 1 parameter set. Since we cannot see the inflection points for $t = 134$, we can exaggerate the error weight t so that we can see the DFR curve in the interval of interest. Ideally, it should be concave at least within all values of $r < 12{,}323$, since this is the extrapolated value of r for BIKE Level 1.

As we can see in Figure 5, this is not what happens for $t = 151$, 153 and 155, for BGF with 5 iterations. Therefore, considering our results regarding the non-concavity of BGF with 2 and 3 iterations, together with the non-concavity of BGF with 2 to 5 iterations when $t = 151$, we believe that it is not conservative to assume that the DFR curve for BGF is concave. We also tested BGF for levels 3 and 5, observing an analogous behavior for $t = 220$ and $t = 300$, respectively.

The main cause for this behavior appears to be the threshold function that depends on $|\mathbf{s}|$. We conclude that it is not safe to use it for the first, and most important, iteration, but Figure 4 suggests that it might be used in further iterations, since it converges to $(d+1)/2$. Initially, we though that the threshold problem would be fixed by defining a maximum value for τ_0. In our exploratory tests, this indeed make concave DFR curves for exaggerated values of t, but the error correction was negatively affected. Therefore, we leave the problem of finding better thresholds for future work.

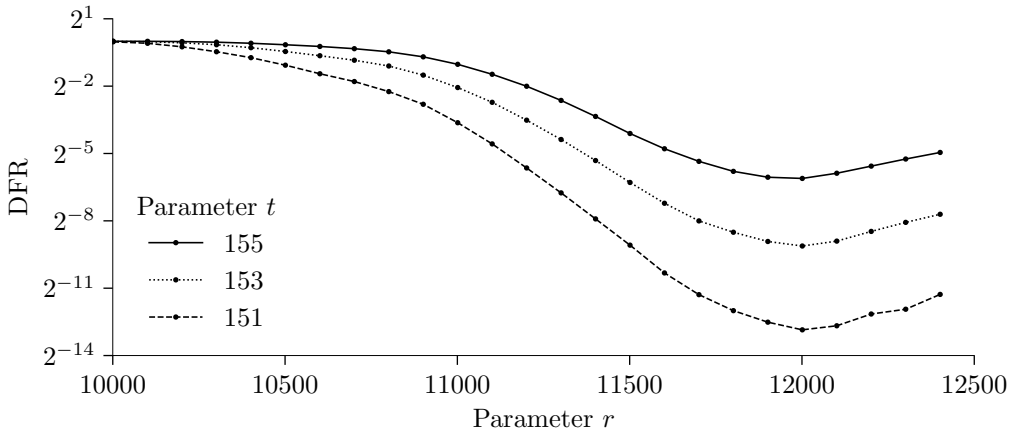

Figure 5: The DFR plot for different values of t considering BIKE Level 1 parameter set.

Our approach to deal with the first iteration is simple: we do not use a simple threshold to flip bits. Instead of starting with a BitFlip iteration, we propose to start with FixFlip, a new type of iteration that works by flipping a predetermined number of bits that have the largest corresponding UPC.

5 PickyFix

In this section, we describe a new BIKE decoder called PickyFix. Similar to other iterative decoders for LDPC codes, PickyFix works by performing a sequence of iterations that progressively increases the knowledge of the secret sparse error used for encrypting. However, it differs significantly in how it chooses which bits to flip in its iterations. We begin by defining two new types of auxiliary procedures: the FixFlip and PickyFlip iterations, that are the building blocks of our decoder.

5.1 The FixFlip Auxiliary Iteration

While the majority of previous bit-flip approaches are based on flipping all bits whose UPC counters are above a certain threshold, FixFlip flips a predetermined number of bits, denoted by n_{Flips}, that have the highest UPC counters. The formal description of a full iteration of FixFlip is described as Algorithm 4.

Almost every step of the algorithm is standard for other bit-flipping algorithms. However, despite its simplicity, one has to be careful with line 3 when implementing the FixFlip iteration, In Section 7 we discuss this issue and show how this can be done efficiently in linear time on r by using important observations on QC-MDPC parameters.

Algorithm 4 The FixFlip iteration.

1: **procedure** FixFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, n_{\text{Flips}}$)
2: $\quad$ `upc` $\leftarrow \left[\left|\text{supp}\left(\mathbf{s}\right) \cap \text{supp}\left(\mathbf{H}_j^\top\right)\right| \text{ for } j = 1 \text{ to } 2r\right]$ ▷ We need the full UPC array
3: $\quad$ `worst_indexes` $\leftarrow$ list of the indexes j of the n_{Flips} largest values of `upc`
4: $\quad$ **for** j in `worst_indexes` **do**
5: $\quad\quad$ $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$ ▷ Flips coordinate j of $\mathbf{e}$
6: $\quad$ $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
7: $\quad$ **return** $\mathbf{e}, \mathbf{s}$

This iteration is very useful at the start of the decoding process, when there is a lot of uncertainty about the correctness of the bits. We can point two immediate advantages of using FixFlip. First, since the number of flips is fixed, the number of wrong flips done by this iteration is limited. This makes FixFlip useful for small values of r, which is an important property for decoders to be used in Vasseur's [Vas21] DFR extrapolation framework. Second, and most important, FixFlip is immune to the problem of BGF's first threshold that gets larger as r grows, since it does not rely on a generic threshold function that depends only on $|\mathbf{s}|$. In fact, the threshold function for FixFlip depends directly on the UPC values and the target number n_{Flips} of bits to flip.

5.2 The PickyFlip Auxiliary Iteration

PickyFlip is very similar to the BitFlip iteration, except that it uses 2 different threshold: τ_{In} is used to flip zeros to ones and τ_{Out} to flip ones to zeros. In particular, PickyFlip requires that the threshold to flip a zero to a one is greater than or equal to the threshold to flip a one to zero. This makes it picky with respect to the support of $\mathbf{e}$ and explains why we use *in* and *out* to differentiate the thresholds. The iteration is formally described as Algorithm 5.

The power of this iteration is that the weight of $\mathbf{e}$ does not grow too much in one iteration because it is easier to give up on a 1 in the partial error vector $\mathbf{e}$ than to accept one more. Additionally, the effect of one PickyFlip iteration is similar to the sequence of black regret and gray regret steps, for a sufficiently high r. Luckily, because of its similarity with the BitFlip iteration, it can be easily implemented by small adjustments of the code by Drucker et al. [DGK20a] in BIKE Additional Implementation.

5.3 The PickyFix Decoder

We are now ready to define a full decoder, which is described as Algorithm 6. To allow for a direct comparison between PickyFix and BGF, we decided to define it in a similar fashion: the first iteration makes 3 calls of the auxiliary steps, which are then followed by single calls in the next $N_{\text{Iter}} - 1$ iterations.

The threshold τ_{Out} for PickyFix is fixed as $(d+1)/2$ in every iteration, which is the value typically used as the minimum threshold for flipping bits. For the value of τ_{In}, we decided to use the BGF's auxiliary function Thresh which was carefully built by the BIKE team and is sufficiently restrictive for our use case.

FixFlip depends on the following parameters: the number n_{Flips} of flips to be done by FixFlipIter and the number N_{Iter} of iterations. These parameters depend on the security level and significantly impact the decoder's performance. We analyze these parameters in the next section.

Algorithm 5 The PickyFlip iteration.

procedure PickyFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_{\text{In}}, \tau_{\text{Out}}$)
  for $j = 1$ to $2r$ **do**
    $\mathtt{upc}_j = \left|\text{supp}\left(\mathbf{s}\right) \cap \text{supp}\left(\mathbf{H}_j^\top\right)\right|$
    if $\mathbf{e}_j = 0$ and $\mathtt{upc}_j \geq \tau_{\text{In}}$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$
    else if $\mathbf{e}_j = 1$ and $\mathtt{upc}_j \geq \tau_{\text{Out}}$ **then**
      $\mathbf{e}_j \leftarrow \overline{\mathbf{e}_j}$
  $\mathbf{s} \leftarrow \mathbf{z} + \mathbf{e}\mathbf{H}^\top$ ▷ Recomputes the partial syndrome
  return $\mathbf{e}, \mathbf{s}$

Algorithm 6 The PickyFix decoding algorithm.

procedure PickyFix($\mathbf{H} = [\mathbf{H}_0|\mathbf{H}_1], \mathbf{z} = \mathbf{c}\mathbf{H}_0^\top$)
  $\mathbf{e} \leftarrow \mathbf{0} \in \mathbb{F}_2^{2r}$ ▷ Initializes the partial error vector
  $\mathbf{s} \leftarrow \mathbf{z}$ ▷ Initializes the partial syndrome
  for $i = 1$ to N_{Iter} **do**
    ▷ Every time $\mathbf{e}$ and $\mathbf{s}$ are updated, it holds that $\mathbf{s} = \mathbf{z} + \mathbf{e}\mathbf{H}^\top$
    if $i = 1$ **then**
      $\mathbf{e}, \mathbf{s} \leftarrow$ FixFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, n_{\text{Flips}}$)
      $\mathbf{e}, \mathbf{s} \leftarrow$ PickyFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_{\text{In}} = \text{Thresh}(\mathbf{s}), \tau_{\text{Out}} = (d+1)/2$)
      $\mathbf{e}, \mathbf{s} \leftarrow$ PickyFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_{\text{In}} = \text{Thresh}(\mathbf{s}), \tau_{\text{Out}} = (d+1)/2$)
    else
      $\mathbf{e}, \mathbf{s} \leftarrow$ PickyFlipIter($\mathbf{H}, \mathbf{z}, \mathbf{e}, \mathbf{s}, \tau_{\text{In}} = \text{Thresh}(\mathbf{s}), \tau_{\text{Out}} = (d+1)/2$)
  if $\mathbf{e}\mathbf{H}^\top = \mathbf{z}$ **then** ▷ This condition is equivalent to $\mathbf{s} = \mathbf{0}$
    return $\mathbf{e}$
  else
    return $\perp$ ▷ Decoding failure

6 Analysis

The main problem when searching for good parameters $(n_{\text{Flips}}, N_{\text{Iter}})$ is that they are not independent. For example, if n_{Flips} is too small, we may need a large number N_{Iter} of iterations to compensate. To simplify our search, we will take a greedy approach and break the search into two parts.

In this section, first we find good values for n_{Flips} by focusing only on the first iteration and then show that these values indeed yield decoders with a concave DFR curve. Finally, we proceed to evaluate the decoder performance for different number N_{Iter} of iterations.

6.1 Choosing the FixFlip Parameter

Intuitively, the best value of n_{Flips} is the one that minimizes the number of errors left to be corrected by further PickyFlip iterations. Ideally, one could see how each possible value of n_{Flips} affects the DFR curves following the extrapolation framework, and choose the one that has the fastest decay. The problem of this approach is that these experiments are very expensive and could easily take months of computing power.

To deal with this problem, instead of counting decoding failures, we count the average number of uncorrected errors left, which can be estimated with a much smaller sample than what is needed for the DFR estimation. Consider the curves $\text{PickyFix}_1^{n_{\text{Flips}}}(r)$ that represent the average number of errors left after the first iteration of PickyFix when the FixFlip iteration performs n_{Flips} bit flips. Similarly, define the curve $\text{BGF}_1(r)$ as the average number of errors left after the first iteration of BGF.

Figure 6 shows selected curves, where the average number of errors left was obtained by simulations of 10,000 runs. Notice how each PickyFlip curve eventually leaves about 0 errors after the first iteration. Furthermore, we can see that BGF appears to stall its error correction in its first iteration as r increases. In Level 1, BGF even starts to leave more errors for sufficiently large values of r, which is a consequence of the very conservative threshold used in the first iteration that we discuss in Section 4.1.

To obtain the best value of n_{Flips} we used the following criteria: for each security level, select the value n_{Flips} such that

$$\text{PickyFix}_1^{n_{\text{Flips}}}(r) = 0, \tag{1}$$

for the lowest value of r. Furthermore, we restricted the search for n_{Flips} to multiples of 5

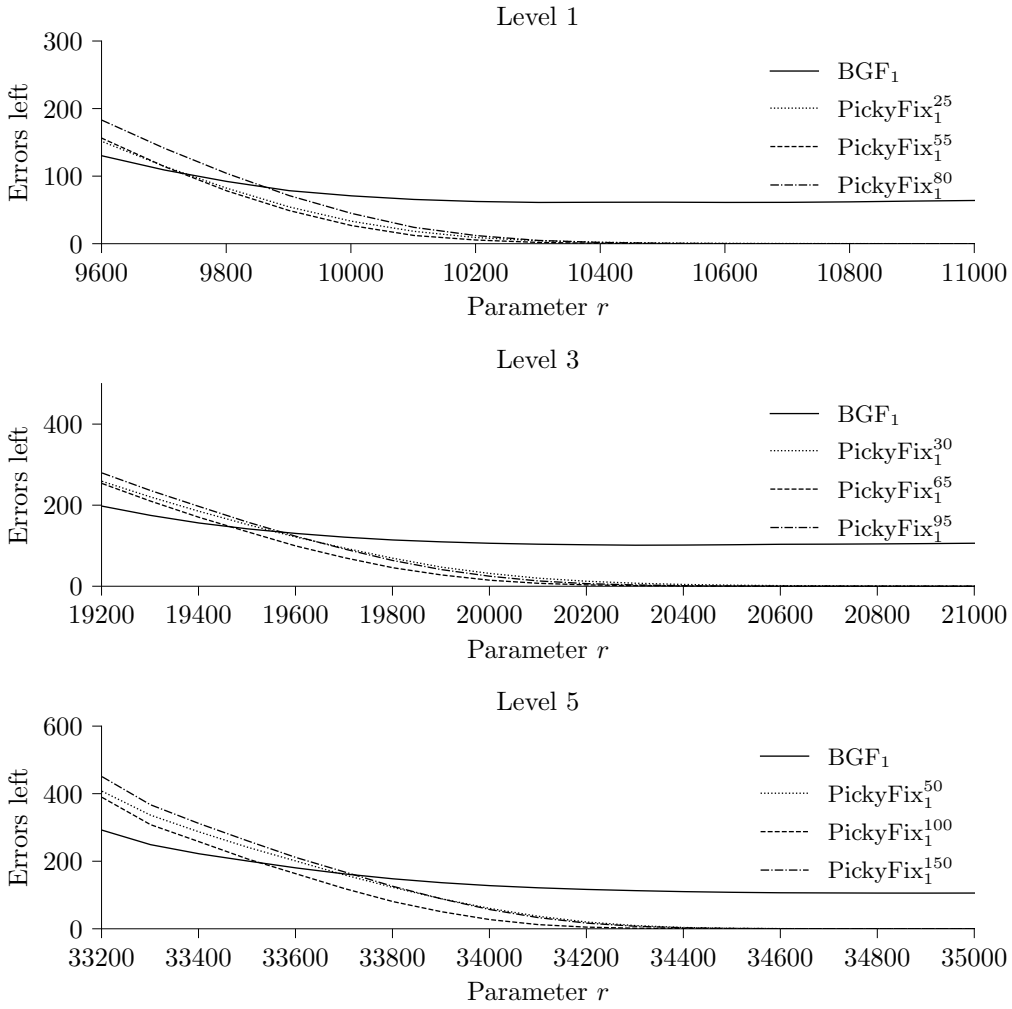

Figure 6: Comparison of the number of uncorrected errors after the first iteration for BGF and FixFlip using different values of n_{Flips}, for the three BIKE parameter sets. The different values of n_{Flips} are indicated by the label format $\mathrm{FixFlip}_1^{n_{\mathrm{Flips}}}$.

to speed up the search. The best values of n_{Flips} obtained for each security level are shown in Table 3, where 10,000 tests were performed to estimate $\mathrm{PickyFix}_1^{n_{\mathrm{Flips}}}(r)$ for each r.

Let us now see how PickyFix behaves with respect to the concavity with an experiment similar to the one done in Section 4.3 for BGF. First notice that we could not use $t = 155$ because PickyFix was much better than BGF's and its DFR quickly got to the point where no failure could be observed in our simulation. Therefore, we had to consider $t = 160$. Figure 7 shows our results for this experiment. We invite the reader to compare this figure with Figure 4.3 and see that, not only PickyFix's DFR appears to be concave in the same interval, but it also outperforms BGF with 5 iterations for a higher value of t. Furthermore, we also tested PickyFix for levels 3 and 5, using $t = 240$ and $t = 330$, respectively, and the DFR curves appear to be concave, unlike the ones for BGF.

Table 3: The best values of n_{Flips} for each security level. Value r_0 denotes the first value of r when Equation 1 is satisfied.

Parameter set	Security level	Value r_0	n_{Flips}	$\text{PickyFix}_1^{n_{\text{Flips}}}(r_0)$	$\text{BGF}_1(r_0)$
BIKE Level 1	128	11,001	55	0.0	63.97
BIKE Level 3	192	21,201	65	0.0	109.06
BIKE Level 5	256	35,001	100	0.0	105.79

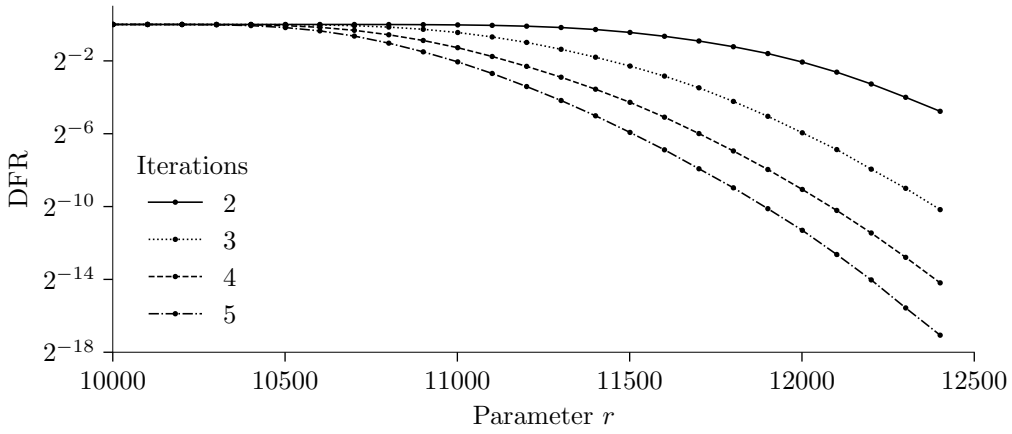

Figure 7: The DFR curves for PickyFix when using 2 to 5 iterations considering the BIKE Level 1 parameter set with $t = 160$.

6.2 Achieving Negligible DFR

Now comes the most important evaluation of PickyFix, which consists of its decoding performance under the extrapolation framework. Our results are shown in Figure 8. The number of tests to determine each DFR estimate was selected to be enough to obtain approximately 1000 failures (at least) for each point and can be found in `data/setup/dfr_experiment.csv`.

Table 5 shows the results for the DFR extrapolation of the curves considered in Figure 8, together with the performance of our constant-time implementations. The extrapolation was done for the last two points (r_A, p_A) and (r_B, p_B) where more than 1000 failures were observed and considered $\alpha = 0.01$ for the Clopper-Pearson method to build the confidence interval for p_A and p_B.

We can see, from Table 5, that even with less than 5 iterations, the extrapolated parameter r for each security level does not differ by much from the parameters proposed by the BIKE team using BGF (Table 1). However, since PickyFix also works with a reduced number of iterations, its performance can be significantly better.

From the results presented in this section, PickyFix looks like a promising decoder for BIKE. However, remember that the FixFlip auxiliary iteration used by PickyFix is inherently more complex than those used by BGF. In the next section, we describe how to efficiently implement PickyFix in constant-time and show that our decoder provides a major speedup over BGF for all security levels.

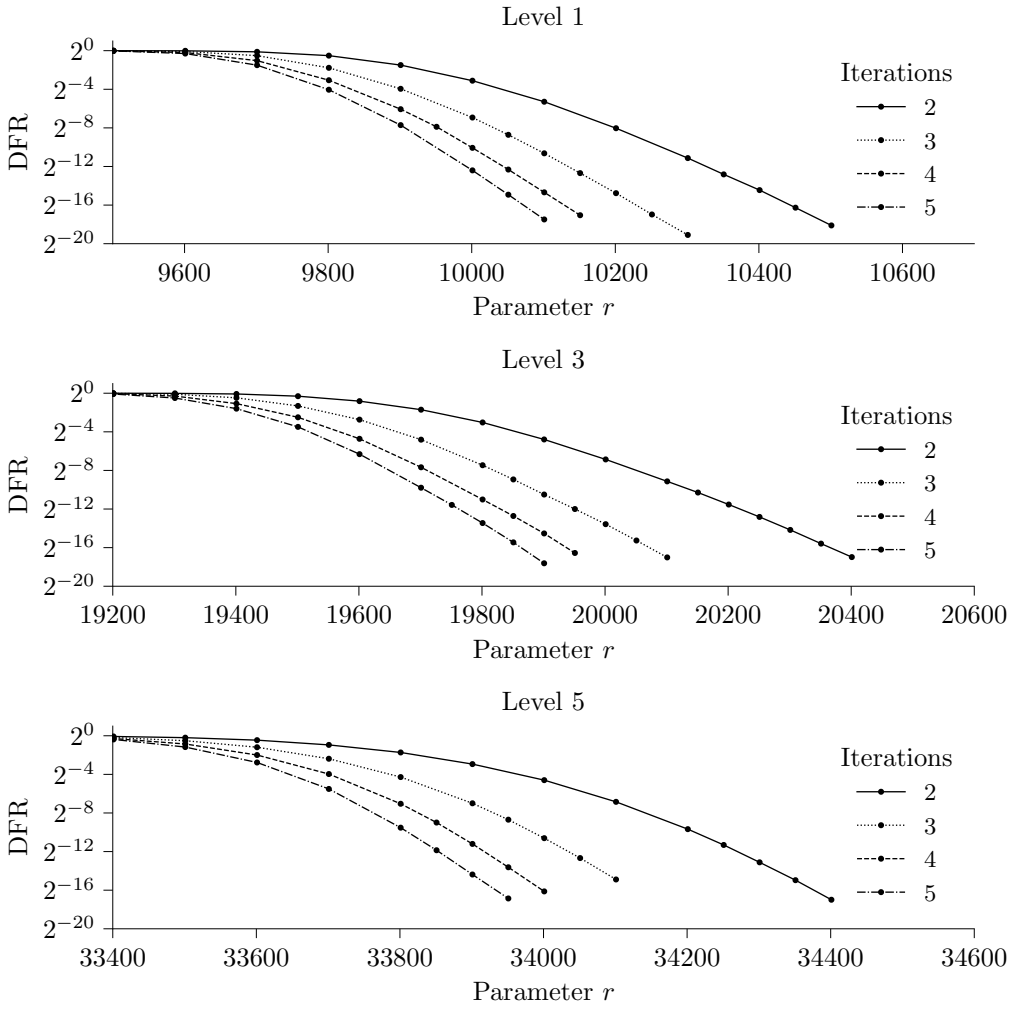

Figure 8: The DFR for PickyFix when using 2 to 5 iterations, considering all security levels.

7 Efficient Implementation in Constant Time

The efficient constant-time implementation proposed by the BIKE team is based on Chou's [Cho16] QcBits with further improvements by Guimarães et al. [GAB19] and Drucker et al. [DGK20b, DG19]. Using these ideas, Drucker et al. [DGK19, DGK20c] proposed the BGF implementation that is the best performing decoder up to this day, which is implemented in BIKE's Additional Implementation [DGK20a].

We based our PickyFix implementation on Drucker's et al. [DGK20a] code, which implements, in constant-time, most of the procedures required for both PickyFlip and FixFlip iterations. This includes, for example, the syndrome and UPC counters computations, and algorithms to flip bits given a threshold.

This section begins with a high-level description on how to adapt Drucker's et al. [DGK20a] implementation to perform the PickyFlip iteration in constant-time. Then we give a more detailed explanation on how to implement the procedures needed by FixFlip that are significantly different from what is used by previous decoders. We end this section

with a performance evaluation of our constant-time implementation, which is available at https://github.com/thalespaiva/pickyfix.

7.1 Implementing the PickyFlip Iteration

Remember that PickyFlip is similar to the BitFlip iteration, except that it uses a different threshold to flip zeros and ones. More specifically, consider the BitFlipIter described in Algorithm 2. Notice how if $\texttt{upc}_j \geq \tau_0$ it inverts $\mathbf{e}_j$, but if $\tau_0 - \delta \leq \texttt{upc}_j < \tau_0$, it updates $\texttt{GrayMask}_j = 1$. BitFlip behavior is then very similar to PickyFix if we let $\tau_0 = \tau_{\text{In}}$ and $\delta = \tau_{\text{In}} - \tau_{\text{Out}}$.

BIKE's efficient implementation of BitFlip is based on QcBits [Cho16], and we implemented PickyFix by reusing their implementation. Since the details of this implementation are already described by Chou [Cho16], we give here only a brief description of how it works.

Suppose we want to flip all bits in $\mathbf{e}$ whose UPC counters are above a threshold τ_{In}. First, all UPC counters are computed in bitsliced form. Since the UPC counters are lower than or equal to $d = w/2$, then $\lceil \log_2(d) \rceil$ slices are enough. Second, the implementation performs a bitsliced subtraction of τ_{In} over all UPC counters. Therefore, the 0 bits in the last slice, which contains the most significant bits, indicate that the UPC was greater than or equal to τ_{In}, and thus the corresponding bit in $\mathbf{e}$ should be flipped.

Notice that PickyFix performs the procedure above two times: one for τ_{In} and other to τ_{Out}. However, the computation of UPC counters, which is the most costly step, is only done once for the two thresholds. The cost of the call is then very similar to the complexity of BitFlipIter.

7.2 Implementing the FixFlip Iteration

Most of the steps needed by the FixFlip algorithm are common to all variants of the original bit-flipping decoder proposed by Gallager [Gal62]. Therefore, we can base our implementation in the most efficient constant-time implementations of QC-MDPC decoders, if we can efficiently implement the sorting step of FixFlip, corresponding to line 3 of Algorithm 4.

Simply put, the main problem we need to solve is: given a list of UPC counters, flip the n_{Flips} bits that have the largest counters. This motivates us to call the set of indexes of entries to be flipped as a FixFlip set, which is formally defined bellow.

Definition 1 (FixFlip set)**.** Consider a list of UPC counters $U = (u_1, \ldots, u_{2r})$. A FixFlip set S with respect to U and n_{Flips} is a set of n_{Flips} indexes such that $u_i \geq u_s$ for all $i \notin S$ and for all $s \in S$.

Notice that, in general, there are more than 1 FixFlip set for the same list of UPC counters. For example, for a list of UPC counters $U = (3, 5, 2, 3, 7, 1, 3, 1)$ and $n_{\text{Flips}} = 4$, then $S_1 = \{1, 2, 4, 5\}$ and $S_2 = \{1, 2, 5, 7\}$ are two valid FixFlip sets. Furthermore, notice that any FixFlip set S for U can be constructed by the threshold $\tau = 3$ and the integer $n_\tau = 1$ by taking every index i whose UPC is strictly greater than τ and also taking n_τ indexes whose UPC is equal to τ. The pair (τ, n_τ) is then called a FixFlip threshold, and is formally defined next.

Definition 2 (FixFlip threshold)**.** Let $U = (u_1, \ldots, u_{2r})$ be a list of UPC counters. A pair (τ, n_τ) is a FixFlip threshold with respect to U and n_{Flips} if, for any FixFlip set S can be partitioned into $S = S_{>\tau} \cup S_{=\tau}$ such that $S_{>\tau} = \{s \in S : u_s > \tau\}$, $S_{=\tau} = \{s \in S : u_s = \tau\}$ and $|S_{=\tau}| = n_\tau$.

This notion helps us to reduce the problem of flipping the bits with the largest UPC values to finding a FixFlip threshold, as shown in Algorithm 7. The idea of the algorithm

Algorithm 7 Algorithm to flip the n_{Flips} entries of $\mathbf{e}$ with largest UPC counters.

```
1:  procedure FlipWorstFitEntries(n_Flips, upc)
2:      τ, n_τ ← FixFlipThreshold(n_Flips, upc)
3:      N_τ ← |{i : upc_i = τ}|
4:      FlipFlagsForThreshold ← Random binary vector of N_τ bits with weight n_τ
5:      η ← 0                                  ▷ Counts the number of bits seen whose upc is τ
6:      for i = 1 to 2r do
7:          if upc_i > τ then
8:              e_i = ē_i
9:          else if upc_i = τ then
10:             η ← η + 1
11:             if FlipFlagsForThreshold_η = 1 then
12:                 e_i = ē_i
13:     return e
```

is to flip all bits whose UPC is above τ, and use the array `FlipFlagsForThreshold` to control which set of n_τ bits should be flipped among all of the N_τ bits whose UPC is τ.

The conditionals in Algorithm 7 can be implemented in constant-time using condition masks. However, there are two aspects that are important to notice when converting the algorithm to a constant-time implementation. The first is that it is not trivial to implement FixFlipThreshold in constant-time. The second is that, to generate the random vector `FlipFlagsForThreshold` of fixed weight in line 4, and to hide the accesses to index η in line 11, we need a tight upper bound on N_τ. In the next two sections, we describe how our constant-time implementation deals with these concerns.

7.2.1 Computing the FixFlip Threshold

The straightforward solution is to use general sorting algorithms, such as quicksort, to sort the indexes based on the corresponding UPC counters' values, and then return the first n_{Flips} indexes. There are two problems with this approach. The first is that the average complexity would be $O(r \log r)$ which would result in an iteration much costlier than that of BGF or BG. The second, and most problematic one, is that the algorithm would not be constant-time and timing attacks would be practical.

Notice that the values of the UPC counters are always in $\{0, \ldots, d\}$, which is a relatively small range, and therefore counting sort is an interesting option that allows for linear sort. The problem with using counting sort in this cryptographic setting is that the constant-time implementation would not be efficient: for every counter, we need to touch all the $d+1$ buckets to avoid cache timing attacks, resulting in $O(wr)$ complexity.

We can do better by analyzing the context in which FixFlip iteration is used. Since the weight t of the error vector is at most $t = 264$, considering security level 5, then it is not necessary to allow for more than 264 flips in each FixFlip iteration. Furthermore, we already saw in Section 6.1 that, in practice, n_{Flips} is typically much lower than t for all security levels, and we can safely assume $n_{\text{Flips}} < 256$. This means that, when performing the counting sort, we only need to count up to 255, since we need only to return the indexes corresponding to the n_{Flips} largest counters. Therefore, 8 bits are needed for each bucket.

Still, even if we can pack 8 buckets into one 64-bit register, we would need to touch all $\lceil (d+1)/8 \rceil$ registers for each counting update. The number of registers would result in $9 \times 2r$ and $18 \times 2r$ operations, considering parameters for levels 1 and 5. But remember that we do not need to count all entries, and we can take what we call the reduced UPC counters approach, which is described next.

Figure 9 shows how the algorithm works in a real decoding instance considering BIKE Level 5. Suppose we are given a list $U = (u_1, \ldots, u_{2r})$ of UPC counters and we want to find

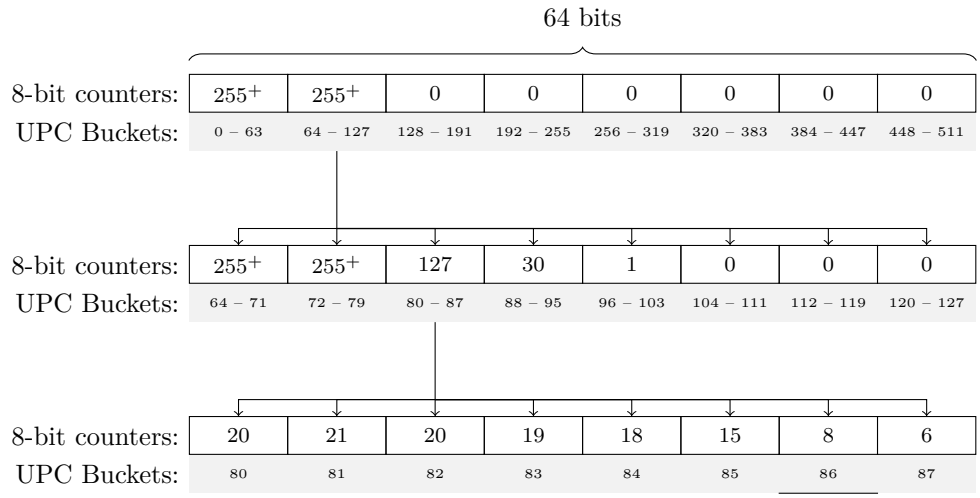

Figure 9: Using 3 levels of partial counting sorts to find the FixFlip threshold for $n_{\text{Flips}} = 40$, considering a real execution of the procedure under BIKE Level 5 parameter set. In this example, the FixFlip threshold corresponds to $\tau = 86$ and $n_\tau = 3$.

the FixFlip threshold for U and $n_{\text{Flips}} < 256$. To show our concrete efficient implementation, we assume the following conditions, that hold in the real world parameters.

1. Each UPC counter $u_i \leq d < 512$.
2. The number of bits to flip is $n_{\text{Flips}} < 256$.

The FixFlip threshold is found in 3 counting steps, and each step uses only 8 buckets. For the first step, each bucket i, where i goes from 0 to 7, corresponds to the UPC counters in the interval $[64i, 64i + 63]$. The algorithm then runs from u_1 to u_{2r} counting the occurrences into the buckets, but with the following rule: the counting is done only in 8 bits, and it should not overflow. That is, the maximum count is 255 for each bucket. Now suppose the resulting counts for each bucket is $[255, 255, 0, 0, 0, 0, 0, 0]$, and consider the case $n_{\text{Flips}} = 40$, just like in Figure 9. Then the bucket where the FixFlip threshold lives must be Bucket 1, since Buckets 3 to 7 do not have any entry, and there are more than n_{Flips} entries in Bucket 1. Using Bucket $b_1 = 1$ selected in this step, the algorithm proceeds to the next step.

In the second step, the algorithm expands Bucket b_1, and the 8 counting buckets are zeroed. Now, each bucket i will count the UPC counters in the interval $[B_2 + 8i, B_2 + 8i + 7]$, where $B_2 = 64b_1$. Again, the algorithm runs through the counters in reversed order until it finds where the FixFlip threshold lives. In the case considered in Figure 9, Bucket 3 is not enough to contain the threshold since it separates at most 31 UPC counters from the rest. Therefore, the search continues using Bucket $b_2 = 2$.

In the third and last step, Bucket b_2 is expanded, and now each counting bucket will correspond to one UPC value. Formally, each Bucket i will count occurrences of the UPC counter $B_3 + i$, where $B_3 = B_2 + 8b_2$. If we consider the search in Figure 9, we can see that it stops at $\tau = 86$, since it has found $6 + 30 + 1 = 37$ UPC values above τ and $n_\tau = 3$ UPC values equal to τ complete the $n_{\text{Flips}} = 40$ bits to be flipped.

Now let us analyze why this algorithm is useful. Since each bucket uses only 8 bits, we can pack all the 8 buckets into a single 64-bits register. Therefore, each update on the counters updates a single register, which avoids the cache-timing attacks. Since 3 rounds are necessary, the threshold is found in about $3 \times 2r$ touches on the counting registers.

Furthermore, let us check that computing the corresponding bucket for an UPC counter is made using constant-time operations. Suppose we want to find the bucket b corresponding to the UPC counter u_i on step ℓ. Then

$$b = \begin{cases} \perp & \text{if } u_i < B_\ell \text{ or } u_i \geq B_\ell + 8^{4-\ell}, \\ \lfloor (u_i - B_\ell)/8^{3-\ell} \rfloor & \text{otherwise.} \end{cases}$$

Both conditions can be evaluated in constant time, since they involve simple unsigned integer comparisons, additions, and the computation of $8^{4-\ell}$ does not involve any secrets. Now for the actual values, if we use 8 bits to represent the buckets, we can let `0xFF` denote the symbol $\perp$. Furthermore, since denominator of the division involving secrets is a power of 8, we can compute $(u_i - B_\ell)/8^{3-\ell}$ in constant time by using a right shift by $3(3-\ell)$ bits, assuming the processor uses a barrel shifter. This observation is particularly useful when considering the vectorized implementation using AVX512 instructions: the bucket computation can be done in parallel for multiple UPC counters, as they involve simple additions, comparisons and right shifts by a fixed amount.

7.2.2 Generating `FlipFlagsForThreshold` and Accessing it in Constant Time

The generation of a random binary vector of a given weight appears frequently in code-based cryptography. For example, both HQC [MAB+18] and BIKE [ABB+21] itself require such a procedure when generating error vectors or secret keys. There is, however, a key difference between our setup and the constant-weight sampling algorithms used by BIKE: FixFlip must hide both the weight n_τ and the size of the vector N_τ.

Let us first see, in Algorithm 8, how the naive Fisher-Yates shuffle works in our case, and then discuss how to make it run in constant-time. We start with a vector of N_τ bits, in which the first n_τ are set to 1 and the rest are set to 0. Then, the algorithm performs n_τ random swaps to shuffle the first n_τ bits of the array. By the end, if each random integer j generated for the swap is unbiased, then each vector of length N_τ and weight n_τ should be generated with uniform probability $1/\binom{N_\tau}{n_\tau}$.

To implement Algorithm 8 in constant-time, we need upper bounds on n_τ, to limit the loops, and on N_τ, to hide the accesses to vector `FlipFlagsForThreshold` when swapping bits in line 9. Notice that, when swapping bits, we only need to hide access to position j, since i is already known in each iteration. Furthermore, notice that we do not use rejection-sampling when selecting the index j because its rejection rate would depend on N_τ. Instead, we use a constant-time modulo reduction of the λ-bit random number, where λ is equal to the security level, to achieve negligible bias.

A trivial upper bound on n_τ is $n_\tau \leq n_{\text{Flips}}$. This allows us to run the loops in lines 3 and 6 in constant time by performing n_{Flips} iterations and using condition masks. Now, to bound N_τ we can focus on the distribution of UPC counters of the wrong bits, that is,

Algorithm 8 Generate a random vector of fixed weight using the Fisher-Yates algorithm.

```
1: procedure GenVectorOfFixedWeight(n_τ, N_τ)
2:     FlipFlagsForThreshold ← 0 ∈ F_2^{N_τ}
3:     for i = 1 to n_τ do
4:         ▷ In the constant-time implementation, bit i is set to 1 only if N_τ ≤ 2κ
5:         FlipFlagsForThreshold_i ← 1
6:     for i = 1 to n_τ do
7:         u ← Random number of λ bits
8:         j ← i + (u  mod (N_τ − i + 1))          ▷ j is a random integer in range i ≤ j ≤ N_τ
9:         Swap bits i and j of FlipFlagsForThreshold
10:    return FlipFlagsForThreshold
```

those that should be flipped. Let U_τ be the random variable that counts the number of UPC counters, among the wrong bits, that are equal to τ. Notice that, when $N_\tau > 2U_\tau$, then flipping bits whose UPC are equal to τ is more likely to result in a wrong flip. Suppose that we find the smallest value κ, in the interval $0 \leq \kappa \leq t$, such that $\Pr(U_\tau > \kappa) \leq 2^{-\lambda}$, where λ is the security level. Then we only care about flipping bits whose UPC are equal to τ in the case when $N_\tau \leq 2\kappa$, as pointed by the comment in line 4 of Algorithm 8.

To find this value κ for each parameter set, we can use Sendrier and Vasseur's [SV19] model for the distributions of UPC counters. Under their model, the UPC counters' distribution for the wrong and right bits are accurately modeled by Binomial distributions with different parameters that are easy to compute. Since we want to consider all possible values of τ, we can search for the smallest κ satisfying the rightmost inequality

$$\Pr(U_\tau > \kappa) \leq \sum_{0 \leq \theta \leq w/2} \Pr(U_\theta > \kappa) \leq 2^{-\lambda},$$

where the distribution of each U_θ is computed using Sendrier and Vasseur's [SV19] model.

Table 4 shows the upper bounds on N_τ that we found for each security level. Our implementation uses an array of 64-bit integers to represent `FlipFlagsForThreshold`, and the total number of 64-bit blocks required for 2κ bits is shown in the last column. Notice that, for security levels 128 and 192, it is possible to simultaneously compute N_τ and the FixFlip threshold, since $2\kappa < 255$. To compute κ, we consider the smallest values of r achieving each security level λ, which are taken from Table 5. This is a conservative approach, since κ gets smaller for higher r within a fixed security level.

Table 4: Upper bounds on N_τ.

Security level λ	r	w	t	κ	$\Pr(U_\tau > \kappa)$	Upper bound 2κ on N_τ	Number of 64-bit blocks
128	12,413	142	134	73	$< 2^{-128.69}$	146	3
192	24,677	206	199	103	$< 2^{-195.54}$	206	4
256	39,019	274	264	130	$< 2^{-259.13}$	260	5

7.3 Performance Evaluation

We now evaluate the decoder with respect to the full decapsulation time[5], when using PickyFix as a subroutine. For this test, we considered the constant-time implementations of BIKE decapsulation using BGF from BIKE Additional Implementation [DGK20a] and our constant-time PickyFix implementation over their code.

The algorithms are implemented in two modes: the portable implementation and the accelerated one using AVX512 instructions. The testing platform consists of an Intel® Xeon™ Gold 5118 CPU at 2.30GHz. Notice that the decoding step is the most important part of the decapsulation. In our setup, the decoding step consists of 90% of the decapsulation, for the portable implementation, and between 80% and 90%, for the AVX512 implementation[6].

Table 5 shows the performance of our constant-time implementation of PickyFix. The basis for the speedup comparison over BGF comes from Table 2, for the corresponding security levels. Notice how PickyFix provides major speedups with respect to BGF for all security levels for one very important reason: it can work with a smaller number of iterations. Even if parameter r suffers a slight increase when using only 2 iterations, between 1% ($\lambda = 256$) and 14% ($\lambda = 128$), this is compensated by speedups from 1.47 to 1.18, correspondingly.

[5]This includes the hashes computations required by the $\text{FO}^{\not\perp}$ transformation.
[6]These number are considering the PickyFix or BGF decoder with 2 iterations.

Table 5: The performance of PickyFix considering the parameters achieving negligible failure rate for each security level. Both the portable and AVX512 implementations were considered, and the speedup is computed with respect to BGF for each level.

Security level	Iterations	r	Portable		AVX512	
			Cycles	Speedup	Cycles	Speedup
128	2	13,829	9,088,162	1.21	1,117,958	1.18
	3	13,109	10,244,537	1.07	1,221,821	1.08
	4	12,739	11,465,955	0.96	1,327,721	1.00
	5	12,413	12,859,593	0.85	1,442,976	0.92
192	2	27,397	25,221,598	1.31	3,196,844	1.29
	3	25,867	28,879,874	1.14	3,577,988	1.15
	4	25,189	32,580,637	1.01	3,935,606	1.05
	5	24,677	36,715,044	0.90	4,350,719	0.95
256	2	41,411	65,388,610	1.45	7,843,855	1.47
	3	39,901	76,892,364	1.23	8,928,026	1.29
	4	39,163	87,393,964	1.09	10,136,052	1.13
	5	39,019	99,706,189	0.95	11,494,770	1.00

8 Conclusion and Future Work

The evidence provided in this paper suggests that PickyFix outperforms BGF both with respect to security and performance. Moreover, we show how PickyFix can be efficiently implemented in constant-time. The only drawback appears to be that the implementation of FixFlip, one of PickyFix's auxiliary iterations, is more involved than that of simple bit-flipping algorithms.

There are several directions one may take to extend this work. It would be interesting to perform a broader exploration of the thresholds used by PickyFlip. For example, to consider looser thresholds for rejecting or accepting ones. On the FixFlip side, notice that we tried to be as general as possible in our implementation. However it may be possible to make it simpler and faster by using the fact that FixFlip is used only in the first iteration. Therefore, one could use statistical analysis to limit the range in which the FixFlip threshold should be searched.

It would be fascinating to see if our implementation of FixFlip can be used to compute better and more complex thresholds. For example, one could use the partial counting of UPC counters to compute thresholds based on the separation of the distributions of UPC for right and wrong bits. On the security side, it is important to understand how PickyFix compares with other decoders in corner cases, such as when using weak keys or decoding near-codeword error patterns [DGK19, SV20b, Vas21]. Finally, it may be interesting to evaluate PickyFix as a decoder for low-density parity-check (LDPC) codes [Gal62].

Acknowledgments

We thank the Continuous Optimization group of Unicamp that provided access to their computer lab to perform the experiments. The lab is supported by FAPESP grant 2018/24293-0. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

References

[ABB+17] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE: Bit flipping key encapsulation, 2017. https://bikesuite.org/files/BIKE.2017.11.30.pdf. Cited on page 1.

[ABB+21] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Carlos Aguilar-Melchor, Rafael Misoczki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit flipping key encapsulation, 2021. https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf. Cited on pages 1, 2, 4, 7, and 20.

[BGG+17] Paulo S. L. M. Barreto, Shay Gueron, Tim Gueneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, and Jean-Pierre Tillich. CAKE: Code-based algorithm for key encapsulation. Cryptology ePrint Archive, Report 2017/757, 2017. https://ia.cr/2017/757. Cited on page 6.

[Cho16] Tung Chou. QcBits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer, 2016. Cited on pages 16 and 17.

[DG19] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *Journal of Cryptographic Engineering*, 9(4):341–357, 2019. Cited on page 16.

[DGK19] Nir Drucker, Shay Gueron, and Dusan Kostic. On constant-time QC-MDPC decoding with negligible failure rate. *IACR Cryptol. ePrint Arch.*, 2019:1289, 2019. Cited on pages 1, 3, 16, and 22.

[DGK20a] Nir Drucker, Shay Gueron, and Dusan Kostic. BIKE Additional Implementation, 2020. https://bikesuite.org/files/round2/add-impl/BIKE_Additional.2020.02.09.zip. Cited on pages 7, 12, 16, and 21.

[DGK20b] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum QC-MDPC cryptography. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 110–127. Springer, 2020. Cited on page 16.

[DGK20c] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In *International Conference on Post-Quantum Cryptography*, pages 35–50. Springer, 2020. Cited on pages 1, 4, 6, and 16.

[FHS+17] T. Fabšič, V. Hromada, P. Stankovski, P. Zajac, Q. Guo, and T. Johansson. A reaction attack on the QC-LDPC McEliece cryptosystem. In *International Workshop on Post-Quantum Cryptography*, pages 51–68. Springer, 2017. Cited on page 1.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999. Cited on page 4.

[GAB19] Antonio Guimarães, Diego F. Aranha, and Edson Borin. Optimized implementation of QC-MDPC code-based cryptography. *Concurrency and Computation: Practice and Experience*, 31(18):e5089, 2019. Cited on page 16.

[Gal62] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962. Cited on pages 2, 3, 6, 17, and 22.

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on MDPC with CCA security using decoding errors. In *22nd Annual International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT)*, 2016. Cited on pages 1 and 4.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017. Cited on pages 1, 3, and 4.

[LJS+16] Carl Löndahl, Thomas Johansson, Masoumeh Koochak Shooshtari, Mahmoud Ahmadian-Attari, and Mohammad Reza Aref. Squaring attacks on McEliece public-key cryptosystems using quasi-cyclic codes of even dimension. *Designs, Codes and Cryptography*, 80(2):359–377, 2016. Cited on page 5.

[MAB+18] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and INSA-CVL Bourges. Hamming quasi-cyclic (HQC). Technical report, Technical report, National Institute of Standards and Technology, 2018. Cited on page 20.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 2069–2073. IEEE, 2013. Cited on pages 1, 2, and 3.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986. Cited on page 1.

[RHHM17] Mélissa Rossi, Mike Hamburg, Michael Hutter, and Mark E Marson. A side-channel assisted cryptanalytic attack against QcBits. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 3–23. Springer, 2017. Cited on page 1.

[Sen11] Nicolas Sendrier. Decoding one out of many. In *International Workshop on Post-Quantum Cryptography*, pages 51–67. Springer, 2011. Cited on page 4.

[SSPB19] Simona Samardjiska, Paolo Santini, Edoardo Persichetti, and Gustavo Banegas. A reaction attack against cryptosystems based on LRPC codes. In *International Conference on Cryptology and Information Security in Latin America*, pages 197–216. Springer, 2019. Cited on page 1.

[SV19] Nicolas Sendrier and Valentin Vasseur. On the decoding failure rate of QC-MDPC bit-flipping decoders. In *International Conference on Post-Quantum Cryptography*, pages 404–416. Springer, 2019. Cited on pages 3 and 21.

[SV20a] Nicolas Sendrier and Valentin Vasseur. About low DFR for QC-MDPC decoding. In *PQCrypto 2020-Post-Quantum Cryptography 11th International Conference*, volume 12100, pages 20–34. Springer, 2020. Cited on pages 1 and 5.

[SV20b] Nicolas Sendrier and Valentin Vasseur. On the existence of weak keys for qc-mdpc decoding. Cryptology ePrint Archive, Report 2020/1232, 2020. https://ia.cr/2020/1232. Cited on page 22.

[Til18] Jean-Pierre Tillich. The decoding failure probability of MDPC codes. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 941–945. IEEE, 2018. Cited on pages 1 and 5.

[TS16] Rodolfo Canto Torres and Nicolas Sendrier. Analysis of information set decoding for a sub-linear error weight. In *Post-Quantum Cryptography*, pages 144–161. Springer, 2016. Cited on page 4.

[Vas21] Valentin Vasseur. QC-MDPC codes DFR and the IND-CCA security of BIKE. Cryptology ePrint Archive, Report 2021/1458, 2021. https://ia.cr/2021/1458. Cited on pages 1, 2, 3, 4, 5, 6, 12, and 22.