Isochronous implementation of the errors-vector generation of BIKE

Nir Drucker[1], Shay Gueron[2,3], and Dusan Kostic[3]

[1]IBM Research, Israel, [2]University of Haifa, Israel, [3]Amazon, USA

Sunday 2nd October, 2022

Abstract. This document describes several considerations that are relevant to an isochronous and constant-time (CT) implementation of the errors-vector generation for the KEM BIKE [2][Ver. 4.2].
Here, the term "constant-time" implementation refers to code that mitigates potential leaks from the micro-architectural features of the processor (e.g., memory access patterns and branches). The term "isochronous" refers to the overall number of steps of an algorithm, which directly affects the overall execution time of its implementation.

Keywords: bit flipping key encapsulation, BIKE, key encapsulation mechanism, rejection sampling, sampling, post-quantum crypto.

1 The timing attack of [5]

BIKE is a key encapsulation mechanism (KEM) and is a submission to the post-quantum cryptography (PQC) standardization project. At this point in time, the national institute of science and technology (NIST) is considering BIKE as a fourth-round KEM candidate for standardization. Hereafter, we refer to version 4.2 [2], which is the Round 3 submission with a few modifications included.

One of the steps of BIKE encapsulation and decapsulation is the generation of an errors-vector. By the definition of [2], the errors-vector is a subset of size t from a set of $2r$ elements. For example, for security Level 1, $t = 134$ and $2r = 2 \cdot 12{,}323 = 24{,}646$. Ideally, this subset needs to be selected, uniformly at random, from the $\binom{2r}{t}$ possible subsets. In practice, BIKE encapsulation starts from a 256-bit seed and thus the number of possible subsets is at most $2^{256} < \binom{2r}{t}$. Moreover, the specification defines a pseudorandom expansion to generate the errors-vector from a seed, using $(e0; e1) \leftarrow H(m)$, where H is modeled as a random oracle. This expansion is specified by Algorithm 3 of [2]. The assupmtion (hope) is that the distribution over the possible subsets is indistinguishable from a uniform random distribution.

Algorithm 3 of [2] is not isochronous. Algorithm 3 therein includes a "while" loop making the number of steps unspecified apriori, and in reality, with a dependence on the seed (technicality it does not specify any bound on the number of steps, but this is not critical to the current discussion). Therefore, an

implementation of Algorithm 3 would not be isochronous, by design. Apparently, isochronous errors-vector generation (EVG) was not perceived as a required property of BIKE. Specifically, BIKE specification explicitly recommends using BIKE with ephemeral keys [2][e.g., on pages 10, 11, 12, 34], where isochronous EVG is assumed to be unnecessary.

Qian Guo et al. [5] describe an attack on BIKE (and HQC [1], but this document discusses only BIKE). This attack leverages the non-isochronous property of Algorithm 3 when using BIKE with a fixed key-pair, despite BIKE's recommendation (see above). Since BIKE's reference code [2] is not a CT implementation, timing attacks, or any other attack on it, are irrelevant. However, they are relevant to the Additional and Optimized implementations package offered by Nir Drucker, Shay Gueron, and Dusan Kostic [3][1] that is referenced in [2].

Before describing a mitigation strategy, we repeat our view, as well as the BIKE specification's guidance: use BIKE only with ephemeral keys (for various reasons)

2 BIKE encapsulation and decapsulation

BIKE encapsulation and decapsulation include an EVG step:

> **errors-vector generation (EVG):** Set t bits in a vector V of length N, at positions chosen (pseudo-)uniformly at random.
>
> (equivalently, choose, uniformly at random, a subset with cardinality t from the $\binom{N}{t}$ possible subsets with cardinality t)

In the encapsulation step, the sender generates the vector by expanding a truly random seed called the message m[2]. The recipient then performs the "re-encryption" step that repeats EVG from the seed m' recovered by the decoder to generate errors-vector e''. If e' is equal to e'' the resulting shared secret is computed with the recovered message m'. Otherwise, it is computed with a dummy random string which is part of the private key.

The EVG can be logically divided into two steps:

1. Run an algorithm that selects (pseudo-)uniformly at random, a subset L of t distinct integers from $\{0, 1, ..., N-1\}$.
2. Set t bits in a zeroed array V, at the positions named by L.

[1] The Additional code package [3] is developed and maintained by Nir Drucker, Shay Gueron, and Dusan Kostic (hereafter, DGK) and in particular, DGK are responsible for any feature (or bug) of this package. This package accompanied BIKE specifications since version 1.0 and it is currently updated to BIKE's version 4.2 [2].

[2] In one of our versions [4] of BIKE [2] the message m is concatenated with the public key of the recipient before the expansion, in order to "bind" a session to a specific key pair. In the decapsulation step, the recipient first decodes the received ciphertext to recover the prospective message m' and errors-vector e'.

The first step can be executed by a rejection sampling strategy:

a. Choose a uniform random value v of B bits, where $B = \lceil \log_2(N)) \rceil$,
b. Reject v if $v > N - 1$, and otherwise accept v and add it to the list L.
c. Continue until L contains t distinct values.

Here, the probability to reject a sample (in Step #b) is $p_{reject} = 1 - \frac{N}{2^B}$.

The number of samples needed to successfully finish this algorithm is an unbounded random variable, i.e., the algorithm may (theoretically) never finish. Therefore, we need to set the limit on the number of samples the algorithm will perform in case it keeps failing to produce enough valid indices. Let the maximum number of samples be X, and assume it is pre-defined in the context of the algorithms hereafter[3] . Then we have two ways to implement the algorithm:

Maximum-Samplings-Number (MSN): Generate samples until we have t distinct indices, but stop after X samples despite the outcome. Output an error indicator if the number of distinct values in the list is less than t even after X selections. Here, the number of steps is not fixed (i.e., the algorithm is not isochronous), but the algorithm terminates in a predetermined time frame.

Fixed-Samplings-Number (FSN): Generate exactly X samples. Output an error indicator if the number of distinct values in the list is less than t. This makes the algorithm flow isochronous (assuming that the elements of the implementation run in CT). The choice of X controls the success probability, i.e., the probability of finding at least t distinct values in exactly X steps. Increasing X increases the success probability but, at the same time, increases the run-time of the algorithm. This run-time is (also) affected by how quickly we can generate random values in Step 1a.

Implementation complications. A secure CT and isochronous implementation needs to face two challenges: hide the timing and memory access in both Step 1 and Step 2. These challenges need a carefully-written code to avoid information leaks. If both steps are done in CT, then the FSN version of the algorithm hides the overall-run-time of the algorithm, while the MSN version does not. The implementation difference between the MSN and the FSN variants is minor and it is relatively easy to accommodate both.

3 Mitigating the timing attack of [5].

The obvious mitigation is to use the FSN version of the EVG, with some predetermined value of X. This value does not change the *required* uniform distribution property of the generated errors-vector, for all cases where enough valid and distinct samples are collected in X samples.

[3] In [3], the constant X is named `MAX_RAND_INDICES_T`. We use the notation X for brevity.

Observations:

1. There is no need to modify the BIKE encapsulation of [3] from MSN-EVG to FSN-EVG, because the encapsulation steps until the EVG step and the EVG itself do not depend on the secret key. Thus, leakage from these steps cannot make the encapsulation serve as a random oracle that leaks information about the secret key. At least for Level 1 security, we suggest that a CT implementation of Steps 1 and 2 is needed only to protect leaking the chosen positions of the errors, i.e., we need to defend against a local "spy" process that can detect the memory locations we are accessing.
2. Depending on the situation, the impact of forcing X selections (via FSN-EVG) could be considered relatively small for the encapsulation functionality. In such cases, an FSN-EVG approach for the encapsulation as well as for the decapsulation could reduce the code complexity (and size). Since this does not affect interoperability, this choice is a matter of preference.
3. Any (even slightly) reasonable choice for the value of X has a negligible performance effect on the decapsulation procedure that is dominated by the decoding step. Choosing X: We first point out that the choice of X for FSN-EVG in decapsulation is an engineering consideration, and not a secur ity consideration. Failure to collect at least t distinct positions for the errors-vector would lead to exactly the same (oblivious) behavior as a decoding failure, with no overall-timing difference. This means that we can choose X to be the smallest value that leads to a tolerable failure rate from the engineering viewpoint. For example, once in 2^{-64} runs seems adequate. Note that we refer to probabilistic failures because an adversary that tries to concoct a ciphertext that leads to a failed decapsulation (not via failed decoding) can achieve this by simply sending a random ciphertext.

3.1 Selecting a "good" value for X

We conclude with the following computation task:

> Given parameters N, B, X, and t compute the success probability of a FSN-EVG implementation.

The computations are detailed as follows, in four steps.

Let p_{valid} denote the probability that a B-bit random sample is a valid sample, i.e., the sample is in the range $[0, N-1]$. Then

$$p_{valid} = \frac{N}{2^B}. \tag{1}$$

Let p_1 denote the probability that exactly k out of n random B-bit samples are valid. Then

$$p_1(k, n) = \binom{n}{k} \cdot p_{valid}^k \cdot p_{reject}^{n-k}. \tag{2}$$

Let p_2 denote the probability that among k *valid* samples there are exactly d distinct values ($d \leq k$). Then

$$p_2(d,k) = \binom{N}{d} \sum_{i=0}^{d} (-1)^i \binom{d}{i} \left(\frac{d-i}{N}\right)^k. \tag{3}$$

Let p_3 denote the probability that among k *valid* samples there are *at least* d distinct values ($d \leq k$). Then

$$p_3(d,k) = \sum_{i=d}^{k} p_2(i,k). \tag{4}$$

Let p_4 denote the probability that among n random B-bit samples there are at least d valid and distinct values. Then

$$p_4(d,n) = \sum_{i=d}^{n} p_3(d,i) \cdot p_1(i,n). \tag{5}$$

Finally, the probability to fail to produce an errors-vector in BIKE, i.e., to get less than t distinct error positions given X random B-bit samples is:

$$p(X) = 1 - p_4(t,X). \tag{6}$$

Examples. For BIKE Level 1 we have $N = 24{,}646$, $B = 15$ ($p_0 = N/2^B \sim 0.7521$), and $t = 134$. For $X = 327$, we have $p(X) \leq 2^{-128}$ (this is the smallest value for which this property holds because $p(X) > 2^{-128}$ for $X = 326$). Alternatively, for $X = 271$, $p(X) \leq 2^{-64}$. The following table shows the failing probability for a few choices of X.

The table also shows, for each X, the performance overhead compared to the non-isocronous MSN-EVG implementation. The overhead is represented in number of CPU cycles. The results were measure on and Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz processor. To put the numbers in perspective, the performance overhead introduced in the decapsulation functionality ranges from 0.05 to 0.3% for Level 1, and from 0.03 to 0.2% for Level 3 parameter sets of BIKE (depending on the choice of X). Furthermore, even if one chooses to use the FSN-EVG in encapsulation as well, e.g., for code simplicity, as we do, the overheads are in the range of $0.7 - 4.1\%$ and $0.6 - 4.9\%$ for Level 1 and 3 respectively (depending on the choice of X).

Fail probability	X for L1	Overhead (cycles)	X for L3	Overhead (cycles)
2^{-192}	N/A	N/A	488	10617
2^{-128}	327	3943	N/A	N/A
2^{-96}	300	2621	405	3115
2^{-80}	286	2156	389	2277
2^{-64}	271	1397	373	1460
2^{-48}	255	639	354	1408

4 Summary

If BIKE is used with a fixed key pair, where the attack of [5] is relevant (for an otherwise CT implementation). In such case, the following mitigation is a simple way to avoid the attack: determine a reasonable value for X, and execute FSN-EVG with X samples for the decapsulation.

Note the following property of our proposed fix: it does not change the Known Answer Tests (KSTs) compared to the BIKE [2][Ver. 4.2] specification, which were committed to NIST and posted as part of the submission. In this respect, our proposed fix is interoperable with [2][Ver. 4.2]. We point out that, in general, changes that break interoperability should be avoided as much as possible. Indeed, in our context, breaking interoperability may lead to consequences for libraries and implementations that may be already using [2][Ver. 4.2].

No change is required for encapsulation. The proposed fix requires a simple straightforward code change in [3] and it is already included therein since June 2022. The change has a negligible performance impact on the decapsulation. Due to the insignificant performance impact, we allowed ourselves to choose $X = 327$ for Level 1 and $X = 488$ for Level 3 (although a smaller number can be chosen as well). For simplified code maintenance, we used an FSN-EVG algorithm for both decapsulation and encapsulation, thus, using the same code as the decapsulation code.

References

1. Aguilar Melchor, C., Aragon, N., Bettaieb, S., Lo ic, B., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G.: Hamming Quasi-Cyclic (HQC) (2017), https://pqc-hqc.org/doc/hqc-specification_2017-11-30.pdf
2. Aragon, N., Barreto, P.S.L.M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Güneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.P., Vasseur, V., Zémor, G.: BIKE: Bit Flipping Key Encapsulation (2021), https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf
3. Drucker, N., Gueron, S., Dusan, K.: Additional implementation of BIKE (Bit Flipping Key Encapsulation), commit 40519b8338ebe1f7bcd0efd8419a180642d94aa4. https://github.com/awslabs/bike-kem (2022)
4. Drucker, N., Gueron, S., Kostic, D.: Binding BIKE Errors to a Key Pair. In: Dolev, S., Margalit, O., Pinkas, B., Schwarzmann, A. (eds.) Cyber Security Cryptography and Machine Learning. pp. 275–281. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-78086-9_21
5. Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(3), 223–263 (Jun 2022). https://doi.org/10.46586/tches.v2022.i3.223-263