Binding BIKE errors to a key pair

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

[1]University of Haifa, Israel, [2]Amazon, USA

Abstract. The KEM BIKE is a Round-3 alternative finalist in the NIST Post-Quantum Cryptography project. It uses the $\mathrm{FO}^{\not\perp}$ transformation so that an instantiation with a decoder that has a DFR of 2^{-128} will make it IND-CCA secure. The current BIKE design does not bind the randomness of the ciphertexts (i.e., the error vectors) to a specific public key. We propose to change this design, although currently, there is no attack that leverages this property. This modification can be considered if BIKE is eventually standardized.

Keywords: BIKE, Post-Quantum Cryptography, NIST, QC-MDPC codes, Ciphertext Binding

1 Introduction

Bit Flipping Key Encapsulation (BIKE) [3] is a Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) code-based Key Encapsulation Mechanism (KEM). It is a Round-3 "alternative finalist" in the NIST Post-Quantum Cryptography project [15]. Figure 1 illustrates BIKE's key generation, encapsulation, and decapsulation flows.

BIKE decapsulation depends on a probabilistic algorithm that is called "Decode", which, for every given input, may succeed (and produce m') or fail (and output $\perp$). Steps 1, 2, and 4 of the encapsulation flow, and steps 2,3 of the decapsulation flow realize the Fujisaki-Okamoto transformation $FO^{\not\perp}$ [12]. This transformation is required in a KEM with possible decapsulation failures for achieving IND-CCA security. Reference [8] proves that BIKE is indeed IND-CCA secure if *Decode* has a Decoding Failure Rate (DFR) of 2^{-128}, 2^{-192}, 2^{-256}, for security levels 1, 3, and 5, respectively. BIKE has the following property.

Property 1. Steps 1,2 of the encapsulation are independent of the public key h.

In a multi-user scenario, Property 1 implies that an adversary can select one errors vector (e_0, e_1) and use it to produce multiple ciphertexts C for different public keys. In this context, we mention the IND-CCA KEM schemes FrodoKEM [2], Kyber [17], Saber [6], SIKE [13], that are selected to Round-3 of the NIST PQC Standardization Project [15] (as either "finalists" or "alternative finalists"). The encapsulation procedures of these KEMs blend the public key value with the randomness. This binds the randomness used for the encapsulation to the session keys (private/public key pair). BIKE [3] and NTRU-Prime [4]

$\underline{(sk, \sigma, h) \xleftarrow{\$} \texttt{Keygen}()}$

1. Generate $\sigma \xleftarrow{\$} \{0,1\}^{256}$
2. $sk = (h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ with $wt(h_0) = wt(h_1) = w$ odd
3. $h = h_1 h_0^{-1}$
4. Return (sk, σ, h)

$\underline{(C, K) \xleftarrow{\$} \texttt{Encaps}(h)}$

1. Generate a message $m \xleftarrow{\$} \mathcal{M}$
2. Compute error vectors $(e_0, e_1) = \mathbf{H}(m)$ with $wt(e_0, e_1) = t$ and $e_0, e_1 \in \mathcal{R}$.
3. Compute the ciphertext $C = (c_0, c_1) = (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
4. Compute the shared key $K = \mathbf{K}(m, C)$

$\underline{m = \texttt{Decaps}(sk, \sigma, h, C)}$

1. $m' = \text{Decode}(sk, C)$ // Or $\perp$ on decoding failure.
2. If $((m' \neq \perp)$ and $(C == \text{ReEncrypt}(m', h)))$ return $\mathbf{K}(m', C)$
3. Else return $\mathbf{K}(\sigma, C)$

Fig. 1. BIKE [3] flows. The block size r and the weights w and t are public parameters of the scheme. $\mathcal{R}$ is the polynomial ring $\mathbb{F}_2[X]/\langle X^r - 1\rangle$. The Hamming weight of an element $v \in \mathcal{R}$ is denoted by $wt(v)$. The $\oplus$ symbol denotes the exclusive-or operation. Uniform random sampling from $\mathcal{R}$ is denoted by $w \xleftarrow{\$} \mathcal{R}$. The key generation outputs a secret key sk, a random seed σ, and a public key h. The input to the encapsulation procedure is the public key h. The output is the ciphertext C and the shared key K. The decapsulation procedure uses the secret key sk, the seed σ, the public key h and the ciphertext C and (always) outputs a shared key K (which is randomized on a decoding failure). $\mathbf{H} : \{0,1\}^{256} \longrightarrow \{0,1\}^{2r}, \mathbf{K} : \{0,1\}^{256+r} \longrightarrow \{0,1\}^{256}, \mathbf{L} : \{0,1\}^{2r} \longrightarrow \{0,1\}^{256}$ are (modeled as) some random oracles with respective output lengths $2r$, 256, 256. They can be instantiated in different ways. $\mathcal{M} = \{0,1\}^{256}$.

(also a Round-3 alternative finalist) use the public key value *only after* the randomness is generated and thus do not possess Property 1 or equivalent. Note that unlike NTRU-Prime, BIKE may encounter decapsulation failures that can lead to reaction attacks [9, 11, 14, 16]. This is potentially exploitable if the scheme's DFR is not negligible. An interesting discussion on the subject can be found on the PQC forum [1] where the discussion ends with:

> [M. Hamburg] "Hashing the public key or its seed is only: A required security feature if your system exhibits decryption failures; and A useful feature to reduce multi-target concerns if your system has any parameter sets aimed at class $\leq$ III."

Examples for the rationale behind the multi-key consideration of Kyber and Frodo are given next. Kyber justifies the discussed binding as follows [5]:

> "This tweak has two effects. First, it makes the KEM contributory; the shared key K does not depend only on input of one of the two parties. The second effect is a multitarget protection. Consider an attacker who searches through many values m to find one that is 'likely' to produce a failure during decryption. Such a decryption failure of a legitimate ciphertext would leak some information about the secret key. [...] hashing pk into $\hat{K}$ ensures that an attacker would not be able to use precomputed values m against multiple targets."

The Frodo team [2] defines a new transformation, namely $FO^{\perp'}$, that is based on $FO^{\perp}$ and states that "following [5], we make the following modifications [..], denoting the resulting transform $FO^{\perp'}$: [..] The computation of r and k also takes the public key pk as input."

Remark 1. Binding the randomness or the ciphertext (without randomness) to the public key is meaningful only if this binding is verified during the decapsulation. In particular, schemes that use the FO [10] transformation, where decapsulation includes re-encryption, verify the binding explicitly (when it exists).

This note discusses the technical considerations that are required for avoiding property 1 in the context of BIKE, with methods to bind the errors vector to a specific public key. We view it as a cheap means to diminish the efficiency of any potential analyses in the multi-key scenario. In this sense, our binding matches BIKE design to that of FrodoKEM, Kyber, Saber, and SIKE.

2 Specific proposals for BIKE

Binding the errors to a specific public key can be done in several ways. Some were mentioned e.g., in [1]: concatenate m to either 1) the public key; 2) the hash digest of the public key; 3) the seed used to generate m (if available). We discuss only options 1 and 2 for BIKE because option 3 is not applicable (in BIKE, the public key is generated from the (secret) private key and not from a publicly known seed).

The function $\mathbf{H} : \{0,1\}^{256} \longrightarrow \{0,1\}^{2r}$ is modeled as a random oracle (see [8] for the details). Its input is a 256 bits seed that $\mathbf{H}$ expands into an errors vector (e_0, e_1). In the current BIKE instantiation, the expander $\mathbf{H}$ is based on AES-CTR PRF, where the input seed plays the role of an AES key. Applying options 1 or 2 above requires another approach. First, using an extractor $f : \{0,1\}^* \longrightarrow \{0,1\}^{256}$ (modeled as a random oracle), to compress the longer input to a 256-bit uniform random string; and subsequently feeding the result into the expander $\mathbf{H}$, as before.

$\underline{(sk, \sigma, h) \xleftarrow{\$} \texttt{Keygen}()}$

1. Generate $\sigma \xleftarrow{\$} \{0,1\}^{256}$
2. $sk = (h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ with $wt(h_0) = wt(h_1) = w$ odd
3. $h = h_1 h_0^{-1}$
4. Return (sk, σ, h)

$\underline{(C, K) \xleftarrow{\$} \texttt{Encaps}(h)}$

1. Generate a message $m \xleftarrow{\$} \mathcal{M}$
2. Compute error vectors $(e_0, e_1) = \mathbf{H}(f_i(m, h))$ with $wt(e_0, e_1) = t$ and $e_0, e_1 \in \mathcal{R}$.
3. Compute the ciphertext $C = (c_0, c_1) = (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$
4. Compute the shared key $K = \mathbf{K}(m, C)$

$\underline{m = \texttt{Decaps}(sk, \sigma, h, C)}$

1. $m' = \text{Decode}(sk, C)$ // Or $\perp$ on decoding failure.
2. If $((m' \neq \perp)$ and $(C == \text{ReEncrypt}(m', h)))$ return $\mathbf{K}(m', C)$
   $\triangleright$ ReEncrypt uses $\mathbf{H}(f_i(m', h))$ instead of $\mathbf{H}(m')$
3. Else return $\mathbf{K}(\sigma, C)$

Fig. 2. Variants of BIKE KEM that bind the errors vector to the public key. The two options are reflected through the function f_i, $i = 1, 2$, as explained in the text. The differences are highlighted in red.

To realize options 1 and 2, we use f_1 and f_2, respectively, as follows

$$f_1 : \mathcal{M} \times \mathcal{PK} \longrightarrow \{0,1\}^{256} \qquad f_2 : \mathcal{M} \times \mathcal{PK} \longrightarrow \{0,1\}^{256}$$
$$(m, pk) \longmapsto H(m \mid\mid pk) \qquad (m, pk) \longmapsto H(m \mid\mid H'(pk))$$

Here, $H, H' : \{0,1\}^* \longrightarrow \{0,1\}^{256}$ are collision-resistant cryptographic hash functions (e.g., SHA256), and $\mathcal{PK}$ is the set of BIKE public keys. With no loss of generality, we assume that $H = H'$. The resulting modified version of BIKE is illustrated in Figure 2.

Remark 2. For completeness, we mention the following two obvious options for f and explain why we do not recommend them for BIKE.

1. Pad the public key to the nearest multiple of 256 bits boundary, split the padded string to 256-bit chunks $pk_1, \ldots, pk_q$ (for the appropriate q), and invoke $\mathbf{H}(m \oplus pk_1 \oplus \ldots \oplus pk_n)$ instead of $\mathbf{H}(m)$ as in Figure 1 Step 2. This approach allows an adversary to control the output of $\mathbf{H}$ through the publicly known pk.

2. Concatenating only 256 bits tail (truncation) of pk to m, i.e., calling $\mathbf{H}(m||trunc_{256}(pk))$ instead of $\mathbf{H}(m)$ in Figure 1 Step 2. This requires an assumption that $trunc_{256}(pk)$ is uniformly random (over $\{0,1\}^{256}$). Note that BIKE public keys are not uniformly random strings, for example, their Hamming weight is always even. Therefore, using the public key's tail requires some additional justification.

3 Practical considerations and the BIKE Additional Implementation Package

The general definition of BIKE uses abstract random oracle functions $\mathbf{H}, \mathbf{K}, \mathbf{L}$ [8]. The specification [3] uses a specific instantiation: $\mathbf{H}$ is based on the CTR-AES PRF, while $\mathbf{K}$ and $\mathbf{L}$ use the standard SHA-384 hash function. The git repository [7] holds an "Additional implementation" package for BIKE, and offers a full *constant-time* software suite as follows: a) a portable C (C99) implementation; b) an implementation that leverages the AVX2 architecture features, written in C (with C intrinsics for AVX2 functions); c) an implementation that leverages the AVX512 architecture features, written in C (with C intrinsics for AVX512 functions).

The AVX512 implementation can also be compiled to use the vector PCLMU -LQDQ instruction that is available on the Intel IceLake processors. The package includes testing and invokes the KAT generation utilities provided by NIST. Note that it is a "stand-alone" suite that does not depend on any external library. However, it also includes a compilation option that allows the use of OpenSSL (to consume its AES256 and SHA-384 implementations). The modularity of the code allows for easy selection of different $\mathbf{H}, \mathbf{K}, \mathbf{L}$ options and for the binding function f. For example, it possible to choose SHA-512 truncated to 384 bits instead of SHA-384, or an arbitrary pseudo-random generator for expanding the (extracted) seed into an errors vector. This code structure makes our build system flexible and therefore it is easy to switch between the current and the proposed instantiation through only a compilation flag only.

The sizes of the BIKE public keys are 1541, 3083, and 5122 bytes for Level-1, 3, 5, respectively. We consider the following two options for instantiating f_1 and f_2, using SHA384 (which is anyway currently used):

- f_1 is the 256 least significant bits of SHA384 hash digest of the input $(m \mid\mid pk)$. Here, the input sizes are 1573, 3115, and 5154 bytes require 13, 25, and 41 invocations of the SHA-384 update functions, respectively.
- f_2 and H are the 256 least significant bits of the SHA384 hash digest of the input. When the input to H is pk, the numbers of invocations of the SHA384 update function are 13, 25, and 41, respectively. The function f_2 invokes the SHA384 update function only once (because the input is of length 64 bytes).

We see that computing f_2 requires one additional invocation of the SHA384 update function compared to f_1. However, the impact of this difference on the overall performance of BIKE is negligible. The advantage of using f_2 is that the encapsulator can choose to compute $H(pk)$ only once and reuse the output. This is valuable in protocols that would use BIKE with static keys. By contrast, using f_1 is more efficient for protocols that use BIKE with ephemeral keys as recommended for BIKE [3] ("BIKE is primarily designed to be used in synchronous communication protocols (e.g. TLS) with ephemeral keys"). For such usages, we recommend the use of f_1. However, for static keys we recommend f_2 because there is no performance cost.

	AVX2 Before	AVX2 After	**SlowDown**	AVX512 Before	AVX512 After	**SlowDown**
Encaps L1	124	143	**1.153**	105	121	**1.152**
Decaps L1	2634	2652	**1.007**	1197	1213	**1.013**
Encaps L3	296	325	**1.098**	237	265	**1.118**
Decaps L3	7988	8017	**1.003**	3480	3509	**1.008**

Table 1. The performance cost of our proposal in 10^3 cycles, when BIKE is used with ephemeral keys. Note that the impact on decapsulation is almost negligible.

We implemented our proposed modification in the Additional implementation of BIKE. This implementation is controlled by the compilation flag BLEND_PK, where the default compilation still follows the current version of BIKE specification [3]. The code modification is small due to the code modularity of our package and affects only the sha.h and sha.c files. Table 1 compares the performance with and without our modification, where we observe a slowdown of up to 15.3% in the encapsulation and up to 1.3% in the decapsulation.

4 Conclusion

We (i.e., the authors of this paper, speaking for themselves and not on behalf of the BIKE team) propose to modify BIKE to a variant that binds the errors vector to the public key. The proposed changes to the encapsulation and decapsulation flows are easy to make, have a low-performance impact, and are already demonstrated in our (Additional) implementation package.

Acknowledgments. This research was supported by: NSF-BSF Grant 2018640; NSF Grant CNS 1906360; The Israel Science Foundation (grant No. 3380/19); The BIU Center for Research in Applied Cryptography and Cyber Security, and the Center for Cyber Law and Policy at the University of Haifa, both in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

1. PQC-forum announcement of NTRU-HRSS-KEM and NTRUEncrypt merger (Dec 2018), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/SrFO_vK3xbI/m/utjUZ9hJDwAJ

2. Alkim, E., Bos, J.W., Ducas, L., Easterbrook, K., LaMacchia, B., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: FrodoKEM Practical quantum-secure key encapsulation from generic lattices (2020), https://frodokem.org/
3. 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 (2020), https://bikesuite.org/files/v4.0/BIKE_Spec.2020.05.03.1.pdf
4. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU Prime: Reducing Attack Surface at Low Cost. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography – SAC 2017. pp. 235–260. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-72565-9_12
5. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. In: 2018 IEEE European Symposium on Security and Privacy (EuroS P). pp. 353–367 (2018)
6. D'Anvers, J.P., Angshuman, K., Sinha Roy, S., Vercauteren, F., Maria Bermudo Mera, J., Van Beirendonck, M., Basso, A.: SABER M-LWR-Based KEM (2020), https://www.esat.kuleuven.be/cosic/pqcrypto/saber/
7. Drucker, N., Gueron, S., Kostic, D.: Additional implementation of BIKE. https://github.com/awslabs/bike-kem (2020)
8. Drucker, N., Gueron, S., Kostic, D., Persichetti, E.: On the Applicability of the Fujisaki-Okamoto Transformation to the BIKE KEM. Tech. Rep. Report 2020/510 (2020), https://eprint.iacr.org/2020/510
9. Eaton, E., Lequesne, M., Parent, A., Sendrier, N.: QC-MDPC: A Timing Attack and a CCA2 KEM. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography. vol. 1, pp. 47–76. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3
10. Fujisaki, E., Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO '99. pp. 537–554. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_34
11. Guo, Q., Johansson, T., Stankovski, P.: A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors, pp. 789–815. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-53887-6_29
12. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A Modular Analysis of the Fujisaki-Okamoto Transformation. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography. pp. 341–371. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_12
13. Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., Hess, B., Jalali, A., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Renes, J., Soukharev, V., Urbanik, D.: SIKE – Supersingular Isogeny Key Encapsulation (2020), https://sike.org/
14. Nilsson, A., Johansson, T., Stankovski Wagner, P.: Error Amplification in Code-based Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems (1), 238–258 (2019). https://doi.org/10.13154/tches.v2019.i1.238-258
15. NIST: Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography (2020), last accessed 30 Sep 2020
16. Santini, P., Battaglioni, M., Chiaraluce, F., Baldi, M.: Analysis of Reaction and Timing Attacks Against Cryptosystems Based on Sparse Parity-Check Codes. In: Baldi, M., Persichetti, E., Santini, P. (eds.) Code-Based Cryptography. Springer

International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25922-8
17. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck M., J., Seiler, G., Stehle, D.: CRYSTALS-KYBER (2020), https://pq-crystals.org/kyber/