Dear BIKE submitters,

I had a look [1] at the reference implementation of BIKE and have the following remarks regarding some (minor) differences that I found compared to the specification. I didn't look at any of the optimized implementations.

- The algorithm 5 ("generate pseudorandom bytes of odd weight") doesn't seem to be the same in the specification and in the implementation. On line 2 in the specification, it is specified that to make the weight odd the first bit is flipped, whereas in sampling.c:93 the bit to be flipped is drawn uniformly at random. It's not obvious to me why the implementation is different nor why this change would be necessary, the specification seems simpler and faster.

- The specification of algorithm 6 seems to be missing the "masking" step that occurs in the implementation. In the specification, it draws uniform 32-bit integers (by drawing 4 bytes from the PRF) and then checks if they are smaller than "len" (which doesn't happen often, given that "len" is equal to "r" or "n", which range between 10163 and 72262, much lower than 2^32). On the contrary, the implementation applies a mask to the uniform 32-bit integer, to retain only the lower "ceil(log2(len))" bits, so that the rejection rate is less than 50% for each integer. The implementation is clearly much more efficient.

By the way, it seems to me that the "bit_scan_reverse" function at utilities.h:76 actually returns "floor(log2(len)) + 1" instead of "ceil(log2(len))", although this shouldn't matter as the length is never a power of two for the proposed parameters.

- The approximated formula for the threshold on page 17 for BIKE-1 and BIKE-2, security level 1 is "ceil(13.530 + 0.0069722 |s|)", but the implementation in defs.h:90 uses 0.0069721 for the second constant. Probably not significant given that |s| is a small integer and the result is rounded to an integer, but it would be nice to clarify. I haven't found similar typos for the other security levels.

- The algorithm 3 requires parameters "S" and "delta". It is mentioned that these depend on "r, w, t", but I couldn't find their values in the specification, nor the rationale for them. I found them as "DELTA_BIT_FLIPPING" and "S_BIT_FLIPPING" in the implementation.

- The description of algorithm 7 ("parallel hashing") has some unclear points to me. From what I understood by re-implementing it to double-check, it first splits the input into 8 slices that each contain "alpha" full 128-byte blocks plus a 111-byte remainder (so that after appending the SHA-384 padding this remainder is extended to a full 128-byte block). It then concatenates the SHA-384 hashes of the 8 the slices with the remainder of the input, and applies SHA-384 on it.

My first comment is that in the specification the remainder "y" is before the hashes "X[7] ... X[0]", whereas in parallel_hash.c:49, the "yx" structure actually contains x before y. From this I understand that the specification of this algorithm denotes bytes in right-to-left order (consistent with the "array" notation), which doesn't seem intuitive.

A second point is that "y" in the specification should have "lrem" bytes (which depends on the input length "la"), but in parallel_hash.c:54, "y" is instead fixed to "s*hbs" bytes, and all these bytes are hashed on line 104 (using "sizeof(yx)"). So in the implementation, "y" is padded with zero bytes to fill "s*hbs" bytes that are hashed, contrary to what is written in the specification.

Best regards,
The files I looked at (both obtained from https://bikesuite.org/) have the following SHA-256 hashes.

da77f33b0f33d704309fc926a2796638ac671a4b82f470a16cce29f9d192407e BIKE.pdf

eec9201859598c7d0bae505651b19b04ef40322445974ee97c06210a322364d3 Reference_Implementation.2018.06.29.zip
Dear BIKE submitters,

I had a closer look at the decapsulation method for BIKE, in particular algorithm 3 of the specification ("one-round bit flipping algorithm").

For the loops at lines 7-10 (loop A) and 13-15 (loop B), I considered the reference implementation to derive the "maximum running time", given that the specification doesn't specify this limit. In the implementation (seedefs.h:150-151) the criterion is that each of these loops stops after at most "n" iterations (for the "n" parameter defined in table 3).

I tested the decapsulation algorithm on a "malformed" cryptogram: instead of using the result of an encapsulation, I filled the cryptogram with uniformly random bytes (it seems that this yields an invalid cryptogram with high probability). It turns out that on this malformed input, the decoding loops A and B don't finish "naturally" but reach the limit of n iterations.

I tested the decapsulation method of BIKE-1 Level1, and on this "uniformly random bytes" input, the reference implementation took around 40 seconds to complete (and return "decoding failure") on my machine, which is much slower than what it takes to decode a valid cryptogram (see table 6 of the specification).

This looks like this could be turned into a denial-of-service attack against the decoder.

This raises several questions.

1) How to mitigate this for BIKE?

After taking a closer look on what happens on the malformed input, it seems that loop A quickly converges to a slightly improved syndrome but without going below a weight of "S". After that, the following iterations don't change anything (the "check" function at line 9 of the specification doesn't find a bit to flip).

I modified the exit criterion of loop A to break as soon as an iteration doesn't change the syndrome. After this "optimization", the decoding completes in around 2 seconds, an order of magnitude faster than the textbook decoding, but still several orders of magnitude slower than on a valid input.

Regarding loop B, I noticed that on the malformed input the error weight quickly increases beyond "t", whereas a properly encapsulated input must have an error weight of exactly "t". Even though the bit-flipping may "unflip" a bit and thereby decrease the error weight, it seems to me that the decoding could stop early and fail if the error weight reaches "t + some small constant". This would likely avoid running n iterations of loop B.

Also, in practice, valid inputs seem to take much less than "n" iterations to be decoded, so the hard limit could be reduced well below "n".

It would be interesting to see how better bounds could be derived (for a given failure probability).

2) Does NIST consider decapsulation of malicious inputs a valid threat model for the evaluation of post-quantum algorithms?

As far as I'm aware, the KATs only test decapsulation of the results of encapsulation.

3) How to evaluate the performance of algorithms that are slower on invalid inputs?
In particular, does NIST plan to evaluate the average/median runtime on valid inputs, the worst-case runtime on valid inputs, the worst-case runtime on malicious/random inputs? Or will constant-time algorithms be required?

Similarly, do benchmark frameworks such as SUPERCOP plan to evaluate decapsulation performance on random inputs?

Best regards,
Guillaume Endignoux