Feasibility and Performance of PQC Algorithms on Microcontrollers

Brian Hession   Jens-Peter Kaps

Cryptographic Engineering Research Group – CERG
ECE Department, George Mason University, Fairfax, VA 22030, U.S.A.
http://cryptography.gmu.edu

Abstract

The eXtended eXternal Benchmarking eXtension (XXBX), which was originally developed to support the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR), is a tool that can measure the performance of cryptographic algorithms on a variety of microcontrollers. We expanded XXBX from supporting hashing algorithms and authenticated ciphers to include benchmarking of key encapsulation methods and signature schemes in order to support the NIST Post-Quantum Cryptography (PQC) standardization process. This paper describes the changes to XXBX which were necessary to support PQC and presents the first results we obtained for a variety of PQC candidates. This is a work in progress and more PQC algorithms will be benchmarked as microcontroller friendly implementations are becoming available and as we are expanding our portfolio of supported microcontrollers.

1 Introduction

With quantum computers developing at an increasing rate, it is important to take into consideration the security implications that may come along with the technological progress. Quantum algorithms will make possible breaking many of the encryptions and algorithms used for key sharing in a reasonable amount of time [1]. An effort to preemptively design and implement quantum resistant security is vital to maintain proper security standards.

There are many algorithms and key sharing protocols proven to be quantum resistant that have already been developed. However, libraries that implement such algorithms focus on x86 architecture and benchmarking. Embedded devices and the Internet of Things (IoT) lack extensive development. In 2018, IoT devices connected to the Internet numbered close to 23.14 billion. By 2025, that number is predicted to be closer to 75.44 billion [2]. Such a large subset of Internet connected devices cannot be left behind during the rise of quantum computing.

Two such libraries implementing quantum resistant cryptography are known as libpqcrypto [3] and the Open Quantum Safe Project (liboqs) [4]. These libraries, however, target x86 and x86_64 based architectures specifically. There are few efforts keeping IoT devices up-to-date [5].

Embedded devices each come with their own strict memory and power constraints making the implementation of such instruction-heavy algorithms a very complicated effort. One such cryptographic library exists for ARM Cortex-M4 architectures known as pqm4 [6]. However, even this library overlooks some of the memory constraints of many devices. There exist tools to help the development, testing, and benchmarking on these specific environments–for example, eXtended eXternal Benchmarking eXtension (XXBX) [7, 8] (shown in Fig. 1) which extends XBX [9] and SUPERCOP [10]. Since quantum computing is developing at an increased rate, it is vital for tools, such as XXBX, to keep up-to-date with the new emerging cryptographic standards.

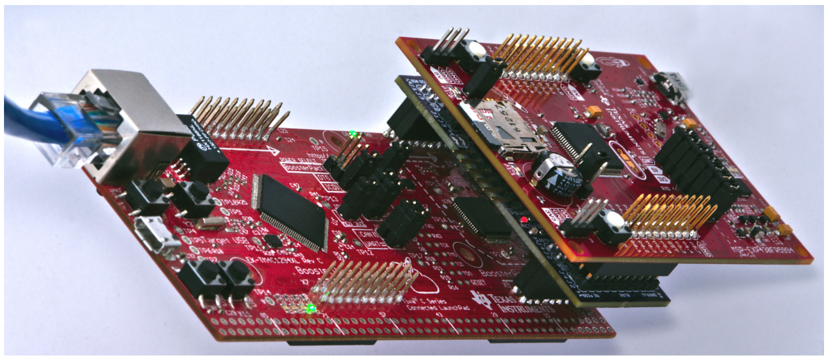

Figure 1: XXBX Setup

2 XXBX Design

XXBX can be broken into four parts: eXternal Benchmarking Software (XBS), eXternal Benchmarking Harness (XBH), eXternal Benchmarking Power (XBP), and eXternal Benchmarking Device (XBD) (see Fig. 2). The XBS is the software running on a PC that handles the cross-compilation of cryptographic algorithms and orchestrates the benchmarking process. It interfaces with the XBH via Ethernet. The XBH acts as the control center and interface between the XBS and XBD. Additionally, it measures the execution time of each algorithm on the XBD and the power consumed with the help of the XBP. The XBP contains a current shunt and an amplifier to enable the XBH to obtain accurate current consumption readings. The XBD is the target device being benchmarked [11].

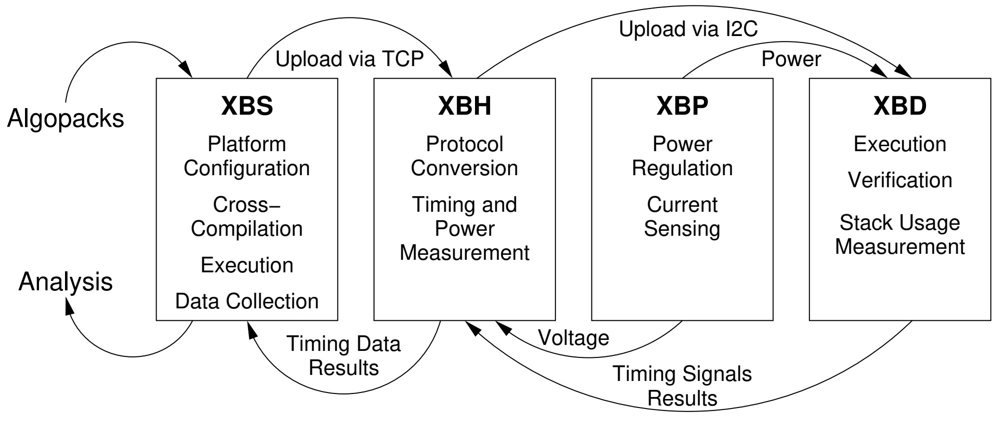

Figure 2: Block Diagram of XXBX Components

2.1 Benchmarking Flow

The XBS cross compiles the cryptographic algorithms that are to be benchmarked using a host of different compiler and linker options to "test applications" for the microcontroller on the selected XBD. Then the benchmarking is performed using the benchmarking flow depicted in Figure 3. The operations of *Encrypt Plaintext*, *Decrypt Ciphertext*, and *Forged Decryption* are based on requirements for benchmarking Authenticated Encryption with Associated Data (AEAD) algorithms. It starts by initiating a timing calibration of the XBH to make sure it can accurately convert the elapsed execution time of an algorithm on the XBD to the number of clock cycles spent on the XBD. It then runs several benchmarking runs on each application. For that, it uploads the application via TCP to the XBH. The XBH forwards the executable to the XBD via I2C and commands the XBD to start running the application. It then sends cipher parameters such as key and message to the XBD followed by a command to start the execution of the cryptographic algorithm. Once received the XBD sends an "execution start" signal back to the XBH upon which the XBH will start measuring the elapsed time. The XBD will execute the uploaded benchmarking test cases and return the results. Along with the results, the XBD will send back the total stack usage. During the execution, the XBH measures the power usage at regular intervals by taking samples from the XBP. The XBD will signal the end of the execution through an "execution end" signal to the XBH. The XBH will gather the power usage and results sent back from the XBD, package them, and send them back to the XBS for analysis. If there are more parameter sets to be tested for this application, the test will continue. Else, the next application will be tested. The XBS will take these results and check for success. If successful, the results are uploaded to a database for further analysis.

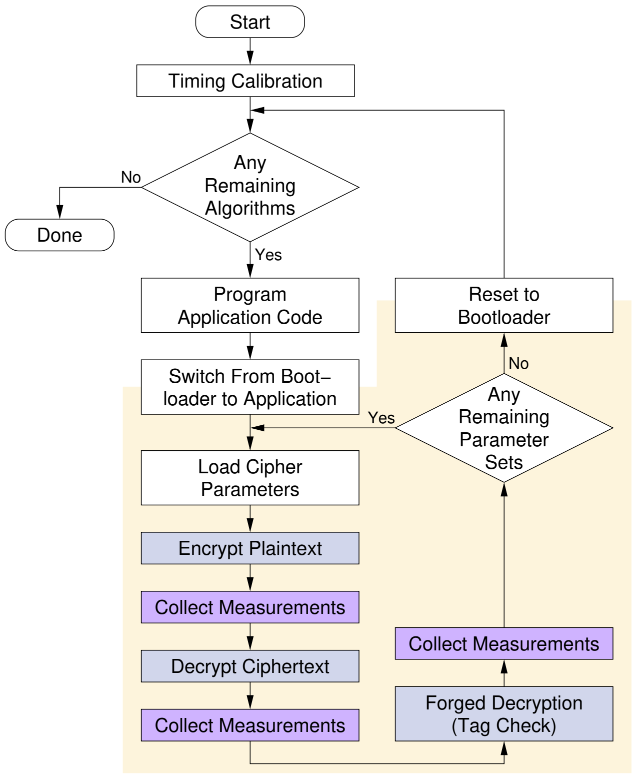

Figure 3: XXBX Execution Flow

2.2 XBD Bootloader

The XBD needs to be loaded with a small bootloader that is able to receive commands and respond to the XBH. The main commands used for execution are the following:

1. *Program Flash Request:* Loads the benchmarking test case application to the ROM.
2. *Timing Calibration Request:* Calibrates the timing differences between the XBH and the XBD to allow for proper timing measurements.
3. *Start Application Request:* Switches the execution from the bootloader to the benchmarking test case application.

2.3 XBD Application

The application can be considered an extension of the XBD bootloader code. These test cases are compiled with a specific cryptographic operation and algorithm. This creates portable code that sits within the ROM that the XBD switches to upon receiving the start application request.

A wrapper that the XBD bootloader understands connects the application and bootloader. To provide general support for all cryptographic operations, there only exists two buffers for execution of tests: parameter buffer and result buffer. The sizes of these buffers are decided at compilation and are unique to the cryptographic algorithm. It is up to the wrapper to compute the correct addresses of the parameters and results.

2.4 XBS

The XBS comprises of a collection of Python scripts. These scripts complete three main functions: compilation, execution, and data recording.

A configuration file sets the cryptographic operation, the algorithm, the specific implementation to test, and the parameters needed to run the test. During compilation, the XBS grabs the specified implementation and the XBD wrapper code needed to execute the operation. Header files following the libcrypto format are generated and the code is compiled along with any dependencies that may be needed. Upon a successful compilation, the database is initialized with the components needed to properly execute the tests. These components include the follow.

- Operation
- Algorithm
- Implementation
- Parameters
- N columns of operation-specific details

During execution, the compiled application is loaded to the XBD for execution. A checksum is performed if the checksum file is present during compilation. The checksum test is essentially a test of sanity. It tests the algorithm for correctness and ensures it follows the expected behavior of the chosen cryptographic operation. Afterwards, the benchmarking is performed. The number of unique tests executed is equal to the number of parameter sets specified. The configuration can specify the number of trials to run per parameter set.

2.5 XBH

The XBH application controls the execution and behavior of the XBD. It receives commands from the XBS, translates them, and performs the specified actions on the XBD.

The device which the XBH code runs on must have a frequency equal to or greater than the device being benchmarked for correct results. Timing calibration is needed between the XBH and XBD to correctly estimate the number of clock cycles required to execute the cryptographic algorithm. When the start execution signal is received from the XBD, the XBH will start timing the execution and gather power usage statistics. This stops when the XBD sends the execution ended signal. At which point, the XBH translates the time taken to clock cycles on the XBD. It then asks for the results and stack usage from the XBD. This all gets packaged and returned to the XBS for analysis.

3 Adding Support for KEMs and Signature Schemes to XXBX

For XXBX to be useful for benchmarking quantum-resistant cryptography, the functionality had to be extended to include key encapsulation methods and signature schemes. This functionality is required in two separate parts of XXBX: XBS and XBD.

3.1 Changes to the XBS

XBS needs to understand the structure and tests needed to support the new functionality. As noted before, XBS initializes the database with the components needed to properly execute the tests. During the execution stage, it grabs these components to forward to the XBH. A translation is needed here to package the data into something that XBD understands. Also, upon return, the data needs to be translated back into something XBS can analyze. This translation should be dependent on the operation but general enough to support many different implementations.

Each trial for KEMs run each of the modes of operation in the following order: key generation, encapsulation, decapsulation, decapsulation failure. For signature schemes, the order is similar: key generation, signing, opening/verifying, forgery detection. The next mode of operation depends on the results of the previous mode. Therefore it is important each mode returns successfully or the trial is cut short, deemed a failure, and XXBX continues on to the next trial.

The parameters to package differ based on the mode of operation. And because there are different modes, an extra variable is needed to specify which mode the XBD should run.

The structure of execution results expected back in return follows a similar design. Because both KEMs and signature schemes depend on the result of the previous mode, these structures need to be kept track of during the life of the trial.

For KEM key generation mode, no parameters are required – just the mode of operation while the results include both the public and secret key. For KEM encapsulation mode, the public key is written to the ROM at the next available block after the application binary. A pointer to this location is provided as a parameter while the results include the session key and ciphertext containing the session key. Lastly, for KEM decapsulation, the secret key is written to ROM (overwriting the public key) and a pointer to its location is provided as an argument along with the ciphertext containing the session key.

Signature schemes are similar. For key generation, no parameters are required and both the public and secret key are returned. For signing, the secret key is written to ROM and a pointer to its location along with the message are passed while the signature is returned. Lastly, for opening/verifying, the public key is written to ROM and a pointer to its location and signature are provided while the results include the verified message.

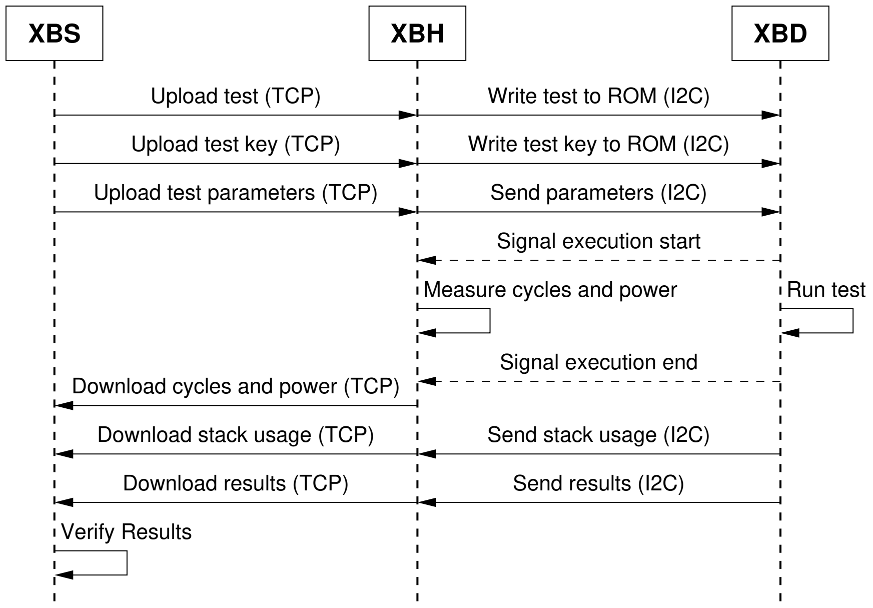

Figure 4: XXBX Public Key Process

3.2 Changes to the XBD

XBD needs to translate the Start Application Request instruction to the intended operation. In order to do so, the data or parameters received by the XBS must match a format expected by the XBD. In turn, the results of the test must be packaged in a format the XBS is expecting.

Prior to unpacking the parameters, the XBD has no idea which mode is being performed. Because of which, the same buffer sizes are allocated regardless of the mode of operation. Therefore, the ROM usage calculated during execution does not accurately reflect the differences between modes and should be considered a general size for the algorithm. The size of the buffers are the largest required of the different modes of operation.

Regardless of success, the length of the returned results does not differ. This is particularly useful for ensuring incorrect decapsulation and detecting signature forgeries on the XBS side.

Unlike the XBS, each execution is independent of the previous.

3.3 XBD Standalone

An additional module was added to the XXBX structure. This XBD standalone mode creates an implementation of the algorithm in an environment without the XXBX overhead. It is a combination of XBD bootloader and XBD application stripped down to the core functions required to execute KEM and Signature operations.

This module automates the build process and allows for testing the algorithm in its purest form for accuracy instead of performance. It also allows for easy debugging with attached debuggers (such as GNUs GDB).

Future work on this module will include an analysis of the stack during execution. In the typical XXBX environment, some of the stack is preserved for XBD application overhead. This module is designed to strip that overhead to its bare minimum and allow debuggers to test the logic of the implementation prior to running performance testing with XXBX. The automated build makes this process simple and extensible.

4 Benchmarking Quantum-Resistant Public Key Cryptographic Algorithms

From the submitted PQC candidates, we attempted to benchmark all algorithms for which the combined size of the session key and the ciphertext do not exceed the RAM and for which the public key and the secret key do not exceed the ROM available on the chosen XBD.

4.1 Target Device

Table 1 shows the microcontroller boards currently supported by XXBX and in purple the controllers that will be supported in the near future. The EK-TM4C123GXL board was chosen to benchmark the PQC algorithms using XXBX. It is currently the best supported ARM Cortex M4F based board by XXBX and represents a rather typical amount of memory, large enough to work with a decent variety of algorithms.

We are in the process of getting the EK-TM4C129EXL which has 4 times the amount of ROM and 8 times the amount of RAM as the EK-TM4C123GXL fully supported and will soon have a full set of results for this board. The next board we will include in XXBX will be the STM32F407G-DISC which is the board used by pqm4 [6]. The amount of memory on the microcontroller is one limiting factor in benchmarking PQC algorithms. In order to expand the amount of PQC algorithms that can be benchmarked we are also planning to support the MSP-EXP432P4111 board which has the most memory of all platforms shown in table 1.

Table 1: Currently Supported XBDs, XBD used in this paper, and XBDs planned for future support

Board	**Manuf.**	**CPU**	**ISA**	**Bus**	**f**	**HW**	**ROM**	**RAM**
MSP-EXP430F5529	TI	MSP430F	MSP430X	16-bit	25 MHz		12kB	10kB
MSP-EXP430FR5994	TI	MSP430FR	MSP430X	16-bit	16 MHz	AES	256kB	8kB
MSP-EXP432P401R	TI	ARM Cortex M4F	ARMv7E-M	32-bit	48 MHz	AES	256kB	64kB
MSP-EXP432P4111	TI	ARM Cortex M4F	ARMv7E-M	32-bit	48 MHz	AES	2048kB	256kB
EK-TM4C123GXL	TI	ARM Cortex M4F	ARMv7E-M	32-bit	80 MHz		256kB	32kB
EK-TM4C129EXL	TI	ARM Cortex M4F	ARMv7E-M	32-bit	120 MHz	AES	1024kB	256kB
NUCLEO-F091RC	STM	ARM Cortex M0	ARMv6-M	32-bit	48 MHz		256kB	32kB
NUCLEO-F103RB	STM	ARM Cortex M3	ARMv7-M	32-bit	72 MHz		128kB	20kB
STM32F407G-DISC1	STM	ARM Cortex M4F	ARMv7E-M	32-bit	168 MHz		1024kB	192kB

4.2 Algorithm Selection

The next "down-selection" came when trying to compile the algorithms. SUPERCOP has a repository of KEM and Signature implementation that fit the XXBX structure. This repository has many of the candidates for the new post-quantum standard. However, some of these algorithms are not capable of being built in embedded environments–particularly because of operating system calls or reliance on libraries such as openssl.

Some additional implementations were pulled from pqm4 to replace those in SUPERCOP. Pqm4 has included some libraries and implementations of common dependencies required by a lot of these algorithms that work with the Cortex-M4 architecture [6]. Recently, pqm4 released round 2 versions of the algorithms they support.

Finally we implemented `_sbrk()` system calls which are needed by all types of `alloc` functions to enable dynamic memory allocation which several algorithms require.

Table 2 list the KEM candidates we are able to benchmark sorted by their respective security levels defined by NIST [12].

Table 2: KEMs than run on XXBX, their key sizes in bytes, and the implementation we ran
(m4v2 – pqm4 library round 2 algorithm, ref – reference implementation, xbdref – modified reference implementation to make it compile on microcontroller)

Algorithm	**Implementation**	**Security Level**	**Type**	**Public Key**	**Secret Key**	**Session Key**	**Ciphertext**
babybear	xbdref	2	Lattice	804	40	32	917
kyber512	m4v2	1	Lattice	736	1632	32	800
lightsaber	m4v2	1	Lattice	672	1568	32	736
newhope512cca	ref	1	Lattice	928	1888	32	1120
ntruhps2048509	m4v2	1	Lattice	699	935	32	699
sikep503	xbdref	1	Isogeny	378	434	16	402
kyber768	m4v2	3	Lattice	1088	2400	32	1152
mamabear	xbdref	4	Lattice	1194	40	32	1307
ntruhps2048677	m4v2	3	Lattice	930	1234	32	930
ntruhrss701	m4v2	3	Lattice	1138	1450	32	1138
saber	m4v2	3	Lattice	992	2304	32	1088
kyber1024	m4v2	5	Lattice	1440	3168	32	1504
newhope1024cca	m4v2	5	Lattice	1824	3680	32	2208
newhope1024cpa	m4v2	5	Lattice	1824	3680	32	2208
papabear	xbdref	5	Lattice	1584	40	32	1697

Unfortunately, most PQC signature algorithms exceed the RAM constraints of our target device. qTesla-I

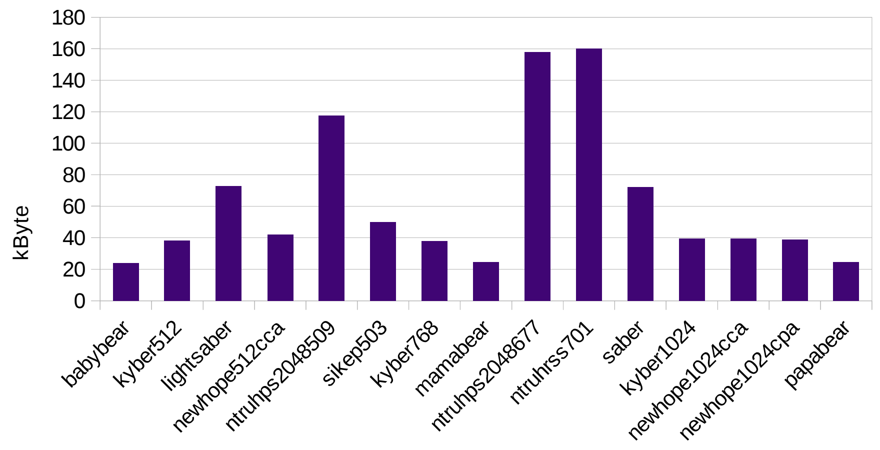

Figure 5: ROM Usage

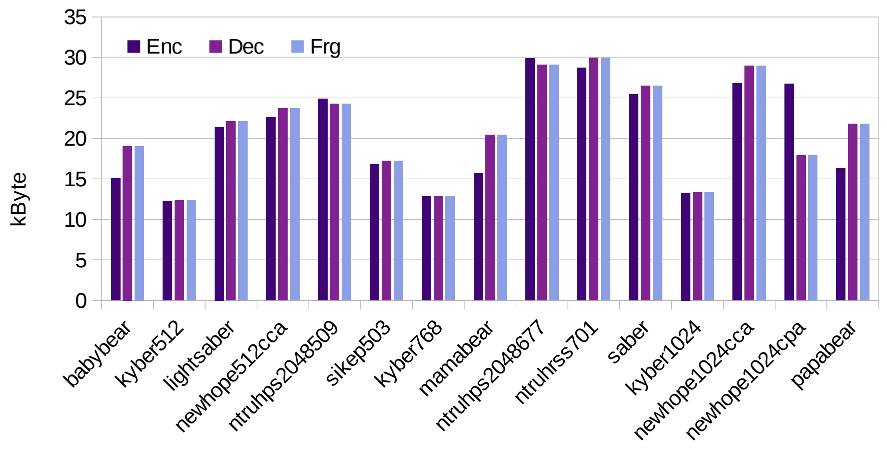

Figure 6: RAM Usage Based on Stack Usage

does work in the XBD-Standalone module but fails with the XBD application overhead. Therefor, we were not able to benchmark any PQC signature algorithm.

An additional hurdle in benchmarking with XXBX is that the XBS network connection to the XBH will timeout during very long calculations on the XBD device. This will typically happen around 40 minutes or so. Due to this, sikep751 and sphincss128shake256 which took more than 1.5 hours could not be benchmarked.

5 Results

XXBX allows us to benchmark the algorithms with respect to ROM usage, RAM usage, speed (in clock cycles), and energy consumption. There are four categories of results: ROM usage, RAM usage, speed (in clock cycles), and energy consumption.

The results for memory usage are shown in Fig. 5 and Fig. 6 for ROM and RAM respectively. ROM usage includes the size of the executable as well as the size for the key written to ROM (see Sect. 3.1). The RAM results are based on the stack usage reported by XXBX. It can be seen that algorithms with larger Ciphertext sizes (see Table 2) are also consuming more RAM, however there are some notable exceptions. Papabear, and both newhopes consume less RAM for their key sizes than others.

The number or clock cycles required varies widely from algorithm to algorithm. The only way to show differences between faster algorithms is by using a logarithmic scale as can be seen in Fig. 7. Sikep530 is the slowest by orders of magnitude. One single encapsulation takes more than 7 minutes, hence its bars leave the graph. Also the three bears are slower than the other algorithms we tested, but they are still reasonably fast. In both cases we used the non optimized reference implementations. The bears seem to take the most time during the noise calculations, in particular the `mac()` function–multiply and accumulate.

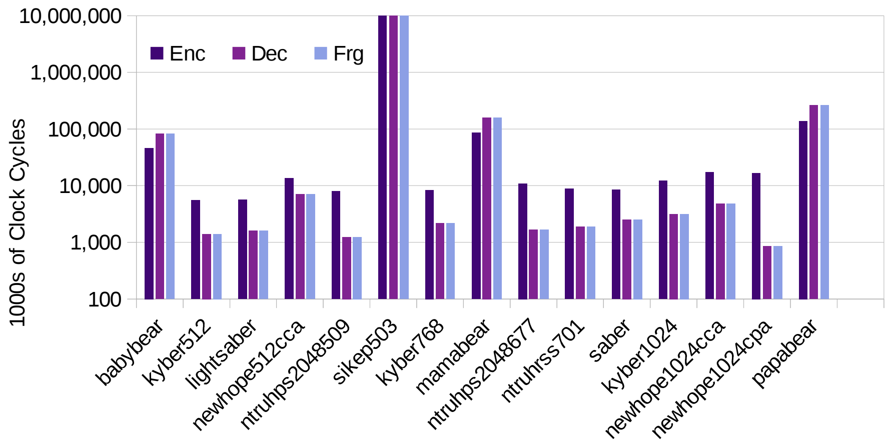

Figure 7: Clock Cycles

Due to the very large variation in number of clock cycles required between algorithms, the amount of energy consumed varies also by several order of magnitudes. Hence, both graphs are very similar.

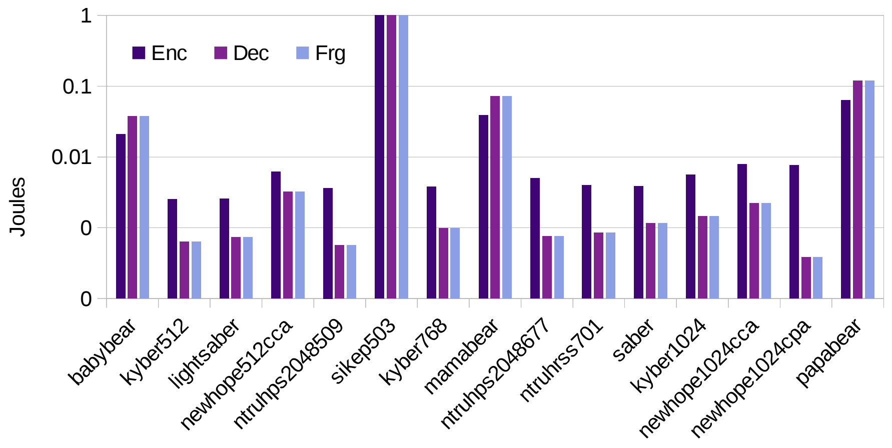

Figure 8: Energy Usage

6 Conclusion

For embedded environments, the strict constraints due to limited memory size and power consumption makes developing IoT devices complicated. If execution time or power consumption are of greatest concern, the Kyber, New Hope, Ntru, and Saber implementations are good candidates for KEM algorithms. However, if memory usage is of greatest concern, the Kyber and Three Bears algorithms are best suited.

Sike takes an incredible amount of time to execute and is not a viable implementation for embedded environments. Sikep503 takes more than 7 minutes while Sikep751 takes almost 40 minutes on the ARM Cortex M4F. Many PQC candidates could not be run on these microcontrollers as their key sizes exceed the available RAM of only 32kByte.

XXBX is simple and adaptable. It will help users shift current cryptographic standards over to quantum-resistant public key cryptography in embedded environments.

References

[1] "CNSA suite and quantum computing FAQ," https://apps.nsa.gov/iaarchive/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm, Jan 2016, identifier: MFQ-U-OO-815099-15.

[2] "Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)," https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2016.

[3] PQCRYPTO ICT-645622, "libpqcrypto," https://libpqcrypto.org/, accessed May 27, 2019.

[4] D. Stebila and M. Mosca, "Post-quantum key exchange for the internet and the open quantum safe project," in *Selected Areas in Cryptography (SAC)*, ser. LNCS, R. Avanzi and H. Heys, Eds., vol. 10532. Springer, Oct 2017, pp. 1–24, https://openquantumsafe.org.

[5] L. Malina, L. Popelova, P. Dzurenda, J. Hajny, and Z. Martinasek, "On feasibility of post-quantum cryptography on small devices," *IFAC-PapersOnLine*, vol. 51, no. 6, pp. 462 – 467, 2018, 15th IFAC Conference on Programmable Devices and Embedded Systems PDeS 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896318308474

[6] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "PQM4: Post-quantum crypto library for the ARM Cortex-M4," https://github.com/mupq/pqm4.

[7] J. Pham and J.-P. Kaps, "eXtended eXternal Benchmarking eXtension (XXBX)," Sep. 2015, presentation at DIAC.

[8] J.-P. Kaps, "eXtended eXternal Benchmarking eXtension (XXBX)," SPEED-B - Software performance enhancement for encryption and decryption, and benchmarking, Oct. 2016, utrecht, Netherlands, invited talk.

[9] C. Wenzel-Benner and J. Gräf, "XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Berlin / Heidelberg: Springer, 2010, pp. 294–305.

[10] D. J. Bernstein and T. Lange, "System for unified performance evaluation related to cryptographic operations and primitives," http://bench.cr.yp.to/supercop.html.

[11] GMU Crytpographic Engineering Research Group, "eXtended eXternal Benchmarking eXtension (XXBX)," https://cryptography.gmu.edu/xxbx/.

[12] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, Dec 2016.