PQCRYPTO
ICT-645622

Horizon 2020

PQCRYPTO

Post-Quantum Cryptography for Long-Term Security

Project number: Horizon 2020 ICT-645622

Small Devices: D1.6 Final Implementations

Due date of deliverable: 28. February 2018
Actual submission date: April 16, 2018

Start date of project: 1. March 2015 — Duration: 3 years

Coordinator:
Technische Universiteit Eindhoven
Email: coordinator@pqcrypto.eu.org
www.pqcrypto.eu.org

Revision 1

Project co-funded by the European Commission within Horizon 2020		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Small Devices: D1.6 Final Implementations

Tim Güneysu, Tobias Oder, Joost Rijneveld, Peter Schwabe, Ko Stoffelen

April 16, 2018
Revision 1

The work described in this report has in part been supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Abstract

This document provides the PQCRYPTO project's final implementations with documentation of build and testing environment, and extensive benchmark results.

Keywords: Post-quantum cryptography, small devices, software implementations, hardware implementations, public-key encryption, public-key signatures, secret-key encryption, secret-key authentication

Contents

1 Introduction

This document describes the final optimized software and hardware implementations of the PQCRYPTO project. The actual code is hosted on GitHub in two repositories:

- https://github.com/mupq/pqm4 containing the **pqm4** post-quantum crypto software library for the ARM Cortex-M4 microcontroller, and
- https://github.com/mupq/pqhw containing the **pqhw** post-quantum crypto hardware library for Xilinx FPGAs.

The actual implementation deliverables are

- the master branch in version `e477e9f956d0511e8053d36bbd8db29c9483df5d` of **pqm4** and
- the master branch in version `dc7075ac183b05d343a70c1209ee975bfb4e6279` of **pqhw**.

In this document we provide the documentation for these implementations, together with extensive benchmarking results. This documentation is (aside from minor modifications for formatting) the documentation also provided in the respective GitHub repositories. Section 2 provides documentation and results of **pqm4** and Section 3 provides documentation and results of **pqhw**.

2 pqm4

Post-quantum crypto library for the ARM Cortex-M4.

2.1 Introduction

The **pqm4** library, benchmarking and testing framework is a result of the PQCRYPTO project funded by the European Commission in the H2020 program.
It currently contains implementations of 8 post-quantum key-encapsulation mechanisms and 2 post-quantum signature schemes targeting the ARM Cortex-M4 family of microcontrollers. The design goals of the library are to offer

- a simple build system that generates an individual static library for each implementation of each scheme, which can simply be linked into any software project;
- automated functional testing on a widely available development board;
- automated generation of test vectors and comparison against output of a reference implementation running host-side (i.e., on the computer the development board is connected to);
- automated benchmarking for speed and stack usage; and
- easy integration of new schemes and implementations into the framework.

2.2 Schemes included in pqm4

Currently **pqm4** contains implementations of the following post-quantum KEMs:

- FrodoKEM-640-cSHAKE
- KINDI-256-3-4-2
- Kyber-768
- NewHope-1024-CCA-KEM
- NTRU-HRSS-KEM-701
- Saber
- SIKE-p571
- Streamlined NTRU Prime 4591761

Currently **pqm4** contains implementations of the following post-quantum signature schemes:

- Dilithium-III
- SPHINCS+-SHAKE256-128s

The schemes were selected according to the following criteria:

- Restrict to NIST round 1 candidates.
- Restrict to schemes and implementations resulting from the PQCRYPTO project.
- Choose parameters targeting NIST security level 3 by default, but
- choose parameters targeting a *higher* security level if there are no level-3 parameters, and
- choose parameters targeting a *lower* security level if level-3 parameters exceed the development board's resources (in particular RAM).
- Restrict to schemes that have at least implementation of one parameter set that does not exceed the development board's resources.

For most of the schemes there are multiple implementations.
The naming scheme for these implementations is as follows:

- `ref`: the reference implementation submitted to NIST,
- `opt`: an optimized implementation in plain C (e.g., the optimized implementation submitted to NIST),
- `m4`: an implementation with Cortex-M4 specific optimizations (typically in assembly).

2.3 Setup/Installation

The testing and benchmarking framework of **pqm4** targets the
STM32F4 Discovery board
featuring an ARM Cortex-M4 CPU, 1MB of Flash, and 192KB of RAM.
Connecting the development to the host computer requires a
mini-USB cable and a USB-TTL converter together with a 2-pin dupont / jumper cable.

2.3.1 Installing the ARM toolchain

The **pqm4** build system assumes that you have the arm-none-eabi toolchain
toolchain installed.

On most Linux systems, the correct toolchain gets installed when you install the `arm-none-eabi-gcc` (or `gcc-arm-none-eabi`) package.
On Linux Mint, be sure to explicitly install `libnewlib-arm-none-eabi` as well (to fix an error relating to `stdint.h`).

2.3.2 Installing stlink

To flash binaries onto the development board, **pqm4** is using stlink.
Depending on your operating system, stlink may be available in your package manager – if not, please
refer to the stlink Github page for instructions on how to compile it from source
(in that case, be careful to use libusb-1.0.0-dev, not libusb-0.1).

2.3.3 Installing pyserial

The host-side Python code requires the pyserial module.
Your package repository might offer `python-serial` or `python-pyserial` directly
(as of writing, this is the case for Ubuntu, Debian and Arch).
Alternatively, this can be easily installed from PyPA by calling `pip install -r requirements.txt`
(or `pip3`, depending on your system).
If you do not have `pip` installed yet, you can typically find it as `python3-pip` using your package manager.

2.3.4 Connecting the board to the host

Connect the board to your host machine using the mini-USB port.
This provides it with power, and allows you to flash binaries onto the board.
It should show up in `lsusb` as `STMicroelectronics ST-LINK/V2`.

If you are using a UART-USB connector that has a PL2303 chip on board (which appears to be the most common),
the driver should be loaded in your kernel by default. If it is not, it is typically called `pl2303`.
On macOS, you will still need to install it (and reboot).
When you plug in the device, it should show up as `Prolific Technology, Inc. PL2303 Serial Port` when you type `lsusb`.

Using dupont / jumper cables, connect the `TX`/`TXD` pin of the USB connector to the `PA3` pin on the board, and connect `RX`/`RXD` to `PA2`.
Depending on your setup, you may also want to connect the `GND` pins.

2.3.5 Downloading pqm4 and libopencm3

Finally, obtain the **pqm4** library and the submodule libopencm3:

```
git clone https://github.com/mupq/pqm4.git
cd pqm4
git submodule init
git submodule update
```

2.4 API documentation

The **pqm4** library uses the NIST API. It is mandated for all included schemes.

KEMs need to define `CRYPTO_SECRETKEYBYTES`, `CRYPTO_PUBLICKEYBYTES`, `CRYPTO_BYTES`, and `CRYPTO_CIPHERTEXTBYTES` and implement

```
int crypto_kem_keypair(unsigned char *pk,
                       unsigned char *sk);
int crypto_kem_enc(unsigned char *ct,
                   unsigned char *ss,
                   const unsigned char *pk);
int crypto_kem_dec(unsigned char *ss,
                   const unsigned char *ct,
                   const unsigned char *sk);
```

Signature schemes need to define `CRYPTO_SECRETKEYBYTES`, `CRYPTO_PUBLICKEYBYTES`, and `CRYPTO_BYTES` and implement

```
int crypto_sign_keypair(unsigned char *pk,
                        unsigned char *sk);
int crypto_sign(unsigned char *sm, unsigned long long *smlen,
                const unsigned char *msg, unsigned long long len,
                const unsigned char *sk);
int crypto_sign_open(unsigned char *m, unsigned long long *mlen,
                     const unsigned char *sm, unsigned long long smlen,
                     const unsigned char *pk);
```

2.5 Running tests and benchmarks

Executing `make` compiles five binaries for each implemenation which can be used to test and benchmark the schemes. For example, for the reference implementation of NewHope-1024-CCA-KEM the following binaries are assembled:

- `bin/crypto_kem_newhope1024cca_ref_test.bin` tests if the scheme works as expected. For KEMs this tests if Alice and Bob derive the same shared key and for signature schemes it tests if a generated signature can be verified correctly. Several failure cases are also checked, see crypto_kem/test.c and crypto_sign/test.c for details.

- `bin/crypto_kem_newhope1024cca_ref_speed.bin` measures the runtime of `crypto_kem_keypair`, `crypto_kem_enc`, and `crypto_kem_dec` for KEMs and `crypto_sign_keypair`, `crypto_sign`, and `crypto_sign_open` for signatures. See crypto_kem/speed.c and crypto_sign/speed.c.

- `bin/crypto_kem_newhope1024cca_ref_stack.bin` measures the stack consumption of each of the procedures involved. The memory allocated outside of the procedures (e.g., public keys, private keys, ciphertexts, signatures) is not included. See crypto_kem/stack.c and crypto_sign/stack.c.

- `bin/crypto_kem_newhope1024cca_ref_testvectors.bin` uses a deterministic random number generator to generate testvectors for the implementation. These can be used to cross-check different implemenatations of the same scheme. See crypto_kem/testvectors.c and crypto_sign/testvectors.c.

- `bin-host/crypto_kem_newhope1024cca_ref_testvectors` uses the same deterministic random number generator to create the testvectors on your host. See crypto_kem/testvectors-host.c and crypto_sign/testvectors-host.c.

The binaries can be flashed to your board using `st-flash`, e.g., `st-flash write bin/crypto_kem_newhope1024cca_ref_test.bin 0x8000000`. To receive the output, run `python3 hostside/host_unidirectional.py`.

The **pqm4** framework automates testing and benchmarking for all schemes using Python3 scripts:

- `python3 test.py`: flashes all test binaries to the boards and checks that no errors occur.
- `python3 testvectors.py`: flashes all testvector binaries to the boards and writes the testvectors to `testvectors/`. Additionally, it executes the reference implementations on your host machine. Afterwards, it checks the testvectors of different implementations of the same scheme for consistency.
- `python3 benchmarks.py`: flashes the stack and speed binaries and writes the results to `benchmarks/stack/` and `benchmarks/speed/`. You may want to execute this several times for certain schemes for which the execution time varies significantly.

In case you don't want to include all schemes, pass a list of schemes you want to include to any of the scripts, e.g., `python3 test.py newhope1024cca sphincs-shake256-128s`.

The benchmark results (in `benchmarks/`) created by
`python3 benchmarks.py` can be automatically converted to the markdown table below using `python3 benchmarks_to_md.py`

2.6 Benchmarks

The tables below list cycle counts and stack usage of the implementations currently included in **pqm4**.
All cycle counts were obtained at 24MHz to avoid wait cycles due to the speed of the memory controller.

2.6.1 Speed Evaluation

2.6.1.1 Key Encapsulation Schemes

scheme	impl.	key generation [cycles]	encapsulation [cycles]	decapsulation [cycles]
frodo640-cshake (10 executions)	opt	AVG: 94,191,951 MIN: 94,191,921 MAX: 94,192,027	AVG: 111,688,861 MIN: 111,688,796 MAX: 111,688,895	AVG: 112,156,317 MIN: 112,156,264 MAX: 112,156,389
kindi256342 (10 executions)	ref	AVG: 22,940,741 MIN: 22,928,176 MAX: 22,947,668	AVG: 29,659,234 MIN: 29,645,532 MAX: 29,674,037	AVG: 37,821,962 MIN: 37,805,302 MAX: 37,840,627
kyber768 (10 executions)	m4	AVG: 1,200,351 MIN: 1,199,831 MAX: 1,200,671	AVG: 1,497,789 MIN: 1,497,296 MAX: 1,498,094	AVG: 1,526,564 MIN: 1,526,070 MAX: 1,526,868
kyber768 (10 executions)	ref	AVG: 1,379,979 MIN: 1,379,339 MAX: 1,380,339	AVG: 1,797,604 MIN: 1,796,996 MAX: 1,797,947	AVG: 1,950,350 MIN: 1,949,742 MAX: 1,950,693
newhope1024cca (10 executions)	ref	AVG: 1,502,435 MIN: 1,502,179 MAX: 1,502,707	AVG: 2,370,157 MIN: 2,369,901 MAX: 2,370,429	AVG: 2,517,215 MIN: 2,516,959 MAX: 2,517,488
newhope1024cca (9 executions)	m4	AVG: 1,246,626 MIN: 1,246,404 MAX: 1,246,772	AVG: 1,966,358 MIN: 1,966,137 MAX: 1,966,505	AVG: 1,977,753 MIN: 1,977,532 MAX: 1,977,899
ntruhrss701 (10 executions)	ref	AVG: 197,262,297 MIN: 197,261,894 MAX: 197,262,845	AVG: 5,166,153 MIN: 5,166,153 MAX: 5,166,155	AVG: 15,069,480 MIN: 15,069,478 MAX: 15,069,485
saber (10 executions)	ref	AVG: 7,122,695 MIN: 7,122,695 MAX: 7,122,695	AVG: 9,470,634 MIN: 9,470,634 MAX: 9,470,634	AVG: 12,303,775 MIN: 12,303,775 MAX: 12,303,775
sikep751 (1 executions)	ref	AVG: 3,508,587,555 MIN: 3,508,587,555 MAX: 3,508,587,555	AVG: 5,685,591,898 MIN: 5,685,591,898 MAX: 5,685,591,898	AVG: 6,109,763,845 MIN: 6,109,763,845 MAX: 6,109,763,845
sntrup4591761 (10 executions)	ref	AVG: 147,543,618 MIN: 147,543,618 MAX: 147,543,618	AVG: 10,631,675 MIN: 10,631,675 MAX: 10,631,675	AVG: 30,641,200 MIN: 30,641,200 MAX: 30,641,200

2.6.1.2 Signature Schemes

scheme	impl.	key generation [cycles]	encapsulation [cycles]	decapsulation [cycles]
dilithium (100 executions)	ref	AVG: 2,888,788 MIN: 2,887,878 MAX: 2,889,666	AVG: 17,318,678 MIN: 5,395,144 MAX: 58,367,745	AVG: 3,225,821 MIN: 3,225,481 MAX: 3,226,288

scheme	impl.	key generation [cycles]	encapsulation [cycles]	decapsulation [cycles]
sphincs-shake256-128s (1 executions)	ref	AVG: 4,433,268,654 MIN: 4,433,268,654 MAX: 4,433,268,654	AVG: 61,562,227,280 MIN: 61,562,227,280 MAX: 61,562,227,280	AVG: 70,943,476 MIN: 70,943,476 MAX: 70,943,476

2.6.2 Stack Usage

2.6.2.1 Key Encapsulation Schemes

scheme	impl.	key generation [bytes]	encapsulation [bytes]	decapsulation [bytes]
frodo640-cshake	opt	36,536	58,328	68,680
kindi256342	ref	10,632	10,736	16,912
kyber768	m4	10,304	13,464	14,624
kyber768	ref	10,304	13,464	14,624
newhope1024cca	m4	11,160	17,456	19,656
newhope1024cca	ref	11,160	17,456	19,656
ntruhrss701	ref	10,024	8,996	10,244
saber	ref	12,616	14,888	15,984
sikep751	ref	11,128	11,672	12,224
sntrup4591761	ref	14,648	10,824	16,176

2.6.2.2 Signature Schemes

scheme	impl.	key generation [bytes]	encapsulation [bytes]	decapsulation [bytes]
dilithium	ref	51,372	87,544	55,752
sphincs-shake256-128s	ref	2,904	3,032	10,768

2.7 Adding new schemes and implementations

The **pqm4** build system is designed to make it very easy to add new schemes and implementations, if these implementations follow the NIST/SUPERCOP API. In the following we consider the example of adding the reference implementation of NewHope-512-CPA-KEM to **pqm4**:

1. Create a subdirectory for the new scheme under `crypto_kem/`; in the following we assume that this subdirectory is called `newhope512cpa`.

2. Create a subdirectory `ref` under `crypto_kem/newhope512cpa/`.
3. Copy all files of the reference implementation into this new subdirectory, except for the file implementing the `randombytes` function (typically `PQCgenKAT_kem.c`).
4. In the subdirectory `crypto_kem/newhope512cpa/ref/` write a Makefile with default target `libpqm4.a`.
   For our example, this Makefile could look as follows:

```
CC     = arm-none-eabi-gcc
CFLAGS = -Wall -Wextra -O3 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
AR     = arm-none-eabi-gcc-ar

OBJECTS= cpapke.o kem.o ntt.o poly.o precomp.o reduce.o verify.o
HEADERS= api.h cpapke.h ntt.h params.h poly.h reduce.h verify.h

libpqm4.a: $(OBJECTS)
  $(AR) rcs $@ $(OBJECTS)

%.o: %.c $(HEADERS)
  $(CC) -I$(INCPATH) $(CFLAGS) -c -o $@ $<
```

Note that this setup easily allows each implementation of each scheme to be built with different compiler flags. Also note the `-I$(INCPATH)` flag. The variable `$(INCPATH)` is provided externally from the **pqm4** build system and provides access to header files defining the `randombytes` function and FIPS202 (Keccak) functions (see below).

1. If the implementation added is a pure C implementation that can also run on the host, then add an additional target called `libpqhost.a` to the Makefile, for example as follows:

```
CC_HOST      = gcc
CFLAGS_HOST  = -Wall -Wextra -O3
AR_HOST      = gcc-ar
OBJECTS_HOST = $(patsubst %.o,%_host.o,$(OBJECTS))

libpqhost.a: $(OBJECTS_HOST)
  $(AR_HOST) rcs $@ $(OBJECTS_HOST)

%_host.o: %.c $(HEADERS)
  $(CC_HOST) -I$(INCPATH) $(CFLAGS_HOST) -c -o $@ $<
```

2. For some schemes you may have a *reference* implementation that exceeds the resource limits
   of the STM32F4 Discovery board. This reference implementation is still useful for **pqm4**, because it can run on the host to generate test vectors for comparison.
   If the implementation you're adding is such a host-side-only reference implementation, place
   a file called `.m4ignore` in the subdirectory containing the implementation.
   In that case the Makefile is not required to contain the `libpqm4` target.

The procedure for adding a signature scheme is the same, except that it starts with creating a new subdirectory under `crypto_sign/`.

2.7.1 Using optimized FIPS202 (Keccak, SHA3, SHAKE)

Many schemes submitted to NIST use SHA-3, SHAKE or cSHAKE for hashing.
This is why **pqm4** comes with highly optimized Keccak code that is accessible
from all KEM and signature implementations.
Functions from the FIPS202 standard (and related publication SP 800-185) are defined in `common/fips202.h` as follows:

```
void shake128_absorb(uint64_t *state,
                     const unsigned char *input, unsigned int inlen);
void shake128_squeezeblocks(unsigned char *output, unsigned long long nblocks,
                            uint64_t *state);
void shake128(unsigned char *output, unsigned long long outlen,
              const unsigned char *input, unsigned long long inlen);

void cshake128_simple_absorb(uint64_t *state,
                             uint16_t cstm,
                             const unsigned char *in, unsigned long long inlen);
void cshake128_simple_squeezeblocks(unsigned char *output, unsigned long long nblocks,
                                    uint64_t *state);
void cshake128_simple(unsigned char *output, unsigned long long outlen,
                      uint16_t cstm,
                      const unsigned char *in, unsigned long long inlen);

void shake256_absorb(uint64_t *state,
                     const unsigned char *input, unsigned int inlen);
void shake256_squeezeblocks(unsigned char *output, unsigned long long nblocks,
                            uint64_t *state);
void shake256(unsigned char *output,
              unsigned long long outlen,
              const unsigned char *input,
              unsigned long long inlen);

void cshake256_simple_absorb(uint64_t *state,
                             uint16_t cstm,
                             const unsigned char *in, unsigned long long inlen);
void cshake256_simple_squeezeblocks(unsigned char *output, unsigned long long nblocks,
                                    uint64_t *state);
void cshake256_simple(unsigned char *output, unsigned long long outlen,
                      uint16_t cstm,
                      const unsigned char *in, unsigned long long inlen);

void sha3_256(unsigned char *output,
```

```
                  const unsigned char *input,  unsigned long long inlen);
  void sha3_512(unsigned char *output,
                  const unsigned char *input,  unsigned long long inlen);
```

Implementations that want to make use of these optimized routines simply include `fips202.h`. The API for `sha3_256` and `sha3_512` follows the
SUPERCOP hash API.
The API for `shake256` and `shake512` is very similar, except that it supports variable-length output.
The SHAKE and cSHAKE functions are also accessible via the absorb-squeezeblocks functions, which offer incremental
output generation (but not incremental input handling).

2.8 License

Different parts of **pqm4** have different licenses. Specifically,

- all files under `common/` are in the public domain;
- all files under `hostside/` are in the public domain;
- all files under `crypto_kem/kyber768/` are in the public domain;
- all files under `crypto_kem/newhope1024cca/` are in the public domain;
- all files under `crypto_kem/ntruhrss701/` are in the public domain;
- all files under `crypto_sign/dilithium/` are in the public domain;
- all files under `crypto_sign/sphincs-shake256-128s/` are in the public domain;
- the files `speed.c`, `stack.c`, `test.c`, `testvectors.c`, `testvectors-host.c` in `crypto_kem/` are in the public domain;
- the files `speed.c`, `stack.c`, `test.c`, `testvectors.c`, and `testvectors-host.c` in `crypto_sign/` are in the public domain
- the files `benchmarks.py`, `benchmarks_to_md.py`, `Makefile`, `README.md`, `test.py`, `testvectors.py`, and `utils.py`
  are in the public domain; and
- the files under `crypto_kem/sikep751/` are under MIT License.
- the files under `crypto_kem/frodo640-cshake/` are under MIT License.
- the files under the submodule directory `libopencm3/` are under LGPL3
- all files under `crypto_kem/sntrup4591761/` are in the public domain;

3 pqhw

Post-quantum crypto implementations for the FPGAs

3.1 Introduction

The **pqhw** implementations are a result of the PQCRYPTO project funded by the European Commission in the H2020 program. Note that these are research oriented implementations and not ready for productive use. It is published under the license contained in the license.rtf file and allows evaluation by academics but no commercial use. Please contact the authors if you intend to use this implementation for other purposes than academic evaluation and verification of our results. The implementations are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

3.2 Schemes included in pqhw

Currently **pqhw** contains implementations of the following post-quantum NIST PQC candidates:

- NewHope-1024

Currently **pqhw** contains implementations of the following post-quantum schemes that are not NIST PQC candidates:

- BLISS

3.3 Setup/Installation

- NewHope was tested with Vivado v2015.3 but should also work with other version of Vivado.
- BLISS was tested with ISE 14.7 but should also work with other version of ISE.
- You can find further information and a copy of the paper and other works on our project website.

3.4 Running tests and benchmarks

- To see NewHope in action run the Test_NewHope.vhd testbench.
- To see BLISS in action run the bliss_sign_then_verify_tb.vhd testbench. Edit the generics to simulate different parameter sets. Some fixed paths might not work (relative is also not an option). Please fix them when you see the error messages.

3.5 Benchmarks

scheme	implementation	LUT	FF	BRAM	DSP	MHz	Cycles
NewHope-1024	server	5,142	4,452	4	2	125	171,124
NewHope-1024	client	4,498	4,635	4	2	117	179,292
BLISS-I	SignHuff_CDT	7,193	6,420	5.5	5	139	15,864
BLISS-I	Sign_BER	8,313	7,932	5	7	142	15,840
BLISS-III	Sign_CDT	6,397	6,179	6.5	5	133	27,547
BLISS-IV	Sign_CDT	6,438	6,198	7	5	135	47,528
BLISS-I	VerifyHuff	5,065	4,312	4	3	166	16,346
BLISS-I	Verify	4,366	3,887	3	3	172	9,607
BLISS-III	Verify	4,298	3,867	3	3	172	9,628

scheme	implementation	LUT	FF	BRAM	DSP	MHz	Cycles
BLISS-IV	Verify	4,356	3,886	3	3	171	9,658

3.6 License

- The License for NewHope can be found in NewHope/license.rtf
- The License for BLISS can be found in BLISS/lattice_processor/license.rtf