PQCRYPTO
ICT-645622

Horizon 2020

PQCRYPTO
Post-Quantum Cryptography for Long-Term Security

Project number: Horizon 2020 ICT-645622

D2.4
Internet: Software library (`libpqcrypto`)

Due date of deliverable: 1. March 2018
Actual submission date: 31. March 2018

WP contributing to the deliverable: WP2

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

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

Revision 1.0

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)	

Internet: Software library (`libpqcrypto`)

Daniel J. Bernstein

31. March 2018
Revision 1.0

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 describes the functionality and use of `libpqcrypto`, WP2's software library for post-quantum cryptography. `libpqcrypto` was publicly released on 14 March 2018 on https://libpqcrypto.org.

Keywords: software, post-quantum cryptography, `libpqcrypto`

Contents

1 Introduction

This document describes `libpqcrypto`, a new cryptographic software library produced by the PQCRYPTO project.

PQCRYPTO, working jointly with many other researchers around the world, submitted 22 proposals to NIST's ongoing post-quantum standardization project.[1] Each submission specifies a family of cryptographic systems, offering various tradeoffs between performance and security. Each submission includes software: a (portable) reference C implementation, and in many cases additional (not necessarily portable) implementations providing better performance (often using assembly language or "intrinsics"). `libpqcrypto` includes software for the following 77 cryptographic systems (50 signature systems and 27 encryption systems) from 19 of the 22 PQCRYPTO submissions:

- BIG QUAKE: `crypto_kem_bigquake{1,3,5}`
- Classic McEliece: `crypto_kem_mceliece{6960119,8192128}`
- CRYSTALS-DILITHIUM: `crypto_sign_dilithium{2,3,4}`
- CRYSTALS-KYBER: `crypto_kem_kyber{512,768,1024}`
- DAGS: `crypto_kem_dags{3,5}`
- FrodoKEM: `crypto_kem_frodokem{640,976}`
- Gui: `crypto_sign_gui{184,312,448}`
- KINDI: `crypto_kem_kindi{256342,256522,512222,512241,512321}`
- LUOV: `crypto_sign_luov{863256,890351,8117404,4849242,6468330,8086399}`
- MQDSS: `crypto_sign_mqdss{48,64}`
- NewHope: `crypto_kem_newhope{512,1024}cca`
- NTRU-HRSS-KEM: `crypto_kem_ntruhrss701`
- NTRU Prime: `crypto_kem_{ntrulpr,sntrup}4591761`
- Picnic: `crypto_sign_picnicl{1,3,5}{fs,ur}`
- qTESLA: `crypto_sign_qtesla{128,192,256}`
- Rainbow: `crypto_sign_rainbow{1a,1b,1c,3b,3c,4a,5c,6a,6b}`
- Ramstake: `crypto_kem_ramstakers{216091,756839}`
- SABER: `crypto_kem_{firesaber,lightsaber,saber}`
- SPHINCS+: `crypto_sign_sphincs{f,s}{128,192,256}{haraka,sha256,shake256}`

[1]NIST issued a formal call for submissions in 2016, set a submission deadline at the end of November 2017, received 82 submissions, and posted 69 "complete and proper" submissions: `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

`libpqcrypto` collects this software into an integrated library, with

- a unified compilation framework,
- an automatic test framework,
- automatic selection of the fastest implementation of each system,
- a unified C interface following the NaCl/TweetNaCl/SUPERCOP/libsodium API,
- a unified Python interface,
- command-line signature/verification/encryption/decryption tools, and
- command-line benchmarking tools.

`libpqcrypto` also integrates some symmetric-crypto software from SUPERCOP, including the AES-256-CTR stream cipher (an OpenSSL wrapper and a separate implementation from Romain Dolbeau), the Salsa20-256 and ChaCha20-256 stream ciphers (implementations from Daniel J. Bernstein, Romain Dolbeau, Martin Goll, Shay Gueron, Ted Krovetz, Tanja Lange, Andrew Moon, Samuel Neves, and Peter Schwabe), the Poly1305 MAC (implementations from Daniel J. Bernstein, Billy Brumley, Andrew Moon, and Peter Schwabe), the SHA-512 hash function (an OpenSSL wrapper, a separate implementation from Daniel J. Bernstein, and a separate implementation from Thomas Pornin via `sphlib`), portions of the Keccak Code Package (from Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer), and the SHAKE256 hash function (a KCP wrapper and implementations from David Leon Gil). For credits regarding the public-key software, see the individual submission packages to NIST.

Beware that the components of `libpqcrypto` vary in licenses. Some parts are in the public domain, but others are not.

2 Security warnings

Most of the primitives (mathematical functions) in `libpqcrypto` are new. **For quantitative and qualitative security analysis, see the individual submission packages, and watch NIST's `pqc-forum` for updates.**

There could be security problems in `libpqcrypto` even if all the proposed primitives achieve their security goals. Most of the software in `libpqcrypto` is new and has not been audited. In particular:

- There could be software bugs that result in the software computing different functions from the proposals, and these differences could destroy security.
- The command-line tools have additional code (input, output, KEM-DEM hybrids, etc.) and have not been audited.
- Some of the software has data-dependent branches and data-dependent array indices, presumably leaking secrets through timings.

New projects in **high-assurance cryptographic software** are working towards engineering a new generation of software with formally verified guarantees of constant-time behavior and full functional correctness. Future updates to `libpqcrypto` will take advantage of this.

3 Installation

3.1 Prerequisites

The following instructions are for Debian/Ubuntu systems. Other modern Linux/BSD/UNIX systems should work with minor adjustments to the instructions. These instructions need the following packages:

- `gcc` and other compiler tools: `apt install build-essential`
- OpenSSL header files: `apt install libssl-dev`
- GMP header files: `apt install libgmp-dev`
- Python 3: `apt install python3`

Check that `df /home/` shows at least 300000 1K-blocks available, and that `df -i /home/` shows at least 30000 inodes free. Currently a typical compile-and-test run uses about 200MB and about 15000 inodes.

3.2 Download, unpack, compile, test, install

In a `root` terminal, create a `libpqcrypto` user:

```
adduser --disabled-password --gecos libpqcrypto libpqcrypto
```

Run a shell as that user:

```
su - libpqcrypto
```

As that user, download and unpack the latest version of `libpqcrypto`:

```
wget -m https://libpqcrypto.org/libpqcrypto-latest-version.txt
version=$(cat libpqcrypto.org/libpqcrypto-latest-version.txt)
wget -m https://libpqcrypto.org/libpqcrypto-$version.tar.gz
tar -xzf libpqcrypto.org/libpqcrypto-$version.tar.gz
cd libpqcrypto-$version
ln -s $HOME link-build
ln -s $HOME link-install
```

Compile, test, and install (this takes time):

```
./do
```

Exit the user shell:

```
exit
```

That's it.

3.3 Options

3.3.1 Remote installation

The download-unpack-compile-test-install process runs entirely from the command line. The process is compatible with the root shell being run under `screen`, and is compatible with this `screen` being run on another machine accessed through `ssh`.

3.3.2 Skipping prerequisites

If OpenSSL and/or GMP are not present, `libpqcrypto` will continue compilation, but it will limit the installation to what it can test. For example, the `ramstake` functions need GMP, and if you compile without GMP then `libpqcrypto` will omit `ramstake`. You can install GMP later and recompile `libpqcrypto`.

3.3.3 Skipping primitives

`./do` will skip a signature system or encryption system if you set the sticky bit on the relevant `crypto_sign` or `crypto_kem` subdirectory. For example, `chmod +t crypto_sign/*/` skips all signature systems; `chmod -t crypto_sign/*/` undoes this. Similar comments apply to lower-level directories for particular implementations.

3.3.4 Compiler options

`./do` tries a *list* of compilers in `compilers/c`, keeping the fastest working implementation of each primitive. Before running `./do` you can edit `compilers/c` to adjust compiler options or to try additional compilers. Beware that each compiler takes time and disk space.

3.3.5 Multi-ABI support

If you put both 32-bit and 64-bit compilers into `compilers/c` then `./do` will produce both 32-bit and 64-bit libraries, available through `lib-x86` and `lib-amd64` (on Intel/AMD CPUs) or `lib-armeabi` and `lib-aarch64` (on ARM CPUs). You should put the 64-bit compilers first so that they are used (if possible) for the command-line tools.

3.4 Future possibilities

3.4.1 Fewer prerequisites

Not many functions in `libpqcrypto` use OpenSSL, and eliminating the OpenSSL dependency will not take much more work. There are already alternative non-OpenSSL implementations for `crypto_stream_aes256ctr` and `crypto_hash_sha512`. OpenSSL is also used for AES-128-CTR in `frodo*`; for SHA-256, SHA-384, and SHA-512 in `gui*` and `rainbow*`; and for SHA-256 in `sphincs*sha256`.

3.4.2 Faster compilation and testing

Some effort will allow compilation and testing to be parallelized on multi-core systems. See also the discussion of shared binaries in Section 8. Speed improvements to `libpqcrypto` will also save time in testing, and increased sharing of internal subroutines will save time in compilation.

3.4.3 Cross-compilation

`libpqcrypto` already has some internal support for cross-compilation. The first stage of `./do`, namely `./build`, only generates `.o` files without running any. The next stage, namely `./test`, links and runs binaries and creates libraries but does not make any new `.o` files.

4 Command-line interface

4.1 Location

To access the `libpqcrypto` command-line tools, add `/home/libpqcrypto/command` to your `PATH`:

```
export PATH=$PATH:/home/libpqcrypto/command
```

You can instead put `/home/libpqcrypto/command/` in front of each command name; but this does not work for the `pq-*-all` wrappers.

4.2 Signature systems

There is a unified interface for all signature systems; these examples use `sphincsf256sha256`. To generate a key pair:

```
pq-keypair-sphincsf256sha256 5>publickey 9>secretkey
```

To sign a message:

```
pq-sign-sphincsf256sha256 <message 8<secretkey >signedmessage
```

To verify a signed message and recover the original message:

```
pq-open-sphincsf256sha256 <signedmessage 4<publickey >message
```

If verification fails, `pq-open-sphincsf256sha256` produces an empty output, prints an error message on `stderr`, and exits 100.

4.3 Encryption systems

There is a unified interface for all encryption systems; these examples use `mceliece8192128`. To generate a key pair:

```
pq-keypair-mceliece8192128 5>publickey 9>secretkey
```

To encrypt a message:

```
pq-encrypt-mceliece8192128 <message 4<publickey >ciphertext
```

To decrypt a ciphertext and recover the original message:

```
pq-decrypt-mceliece8192128 <ciphertext 8<secretkey >message
```

If decryption fails, `pq-decrypt-mceliece8192128` produces an empty output, prints an error message on `stderr`, and exits 100.

4.4 Benchmarking

Run `pq-size-all` to see key sizes etc. (For `picnic*fs`, signature sizes are message-dependent; the maximum possible signature size is reported.) Run `pq-speed-all` to see key-generation times etc. Run `pq-notes-all` for implementation notes. The output formats are subject to change.

5 Python interface

5.1 Location and interface conventions

To access the Python functions provided by `libpqcrypto`, add `/home/libpqcrypto/python` to your `PYTHONPATH`:

```
export PYTHONPATH="/home/libpqcrypto/python${PYTHONPATH+:$PYTHONPATH}"
```

Also insert

```
import pqcrypto
```

into your Python 2 or Python 3 script.

All inputs and outputs are byte strings, the `bytes` type in Python (which is the same as `str` in Python 2 but different in Python 3). Verification failures and decapsulation failures raise exceptions.

5.2 Signature systems

There is a unified interface for all signature systems; these examples use `mqdss64`. To generate a key pair:

```
pk,sk = pqcrypto.hash.mqdss64.keypair()
```

To sign a message `m`:

```
sm = pqcrypto.hash.mqdss64.sign(m,sk)
```

To recover a message from a signed message:

```
m = pqcrypto.hash.mqdss64.open(sm,pk)
```

As a larger example, the following test script signs and then recovers a message under a random key pair:

```
import pqcrypto
sig = pqcrypto.sign.mqdss64
pk,sk = sig.keypair()
m = b"hello world"
sm = sig.sign(m,sk)
assert m == sig.open(sm,pk)
```

5.3 Key-encapsulation mechanisms

There is a unified interface for all KEMs; these examples use `newhope1024cca`. To generate a key pair:

```
pk,sk = pqcrypto.kem.newhope1024cca.keypair()
```

To generate a ciphertext `c` encapsulating a randomly generated session key `k`:

```
c,k = pqcrypto.kem.newhope1024cca.enc(pk)
```

To recover a session key from a ciphertext:

```
k = pqcrypto.kem.newhope1024cca.dec(c,sk)
```

As a larger example, the following test script creates a key pair, creates a ciphertext and session key, and then recovers the session key from the ciphertext:

```
import pqcrypto
kem = pqcrypto.kem.newhope1024cca
pk,sk = kem.keypair()
c,k = kem.enc(pk)
assert k == kem.dec(c,sk)
```

As another example, the following script generates 10000 key pairs and checks that they are all different:

```
import pqcrypto
kem = pqcrypto.kem.newhope1024cca
n = 10000
assert len(set(kem.keypair() for i in range(n))) == n
```

5.4 Lower-level primitives

There are interfaces for various lower-level functions such as stream ciphers (`stream`), one-time authenticators (`onetimeauth`), and hash functions (`hash`). For example, the following test script checks the SHA-512 hash of a string against Python's `hashlib` library:

```
import pqcrypto
m = pqcrypto.randombytes(1234567)
h = pqcrypto.hash.sha512(m)

import hashlib
H = hashlib.sha512()
H.update(m)
assert H.digest() == h
```

The following test script computes and checks an authenticator:

```
import pqcrypto
mac = pqcrypto.onetimeauth.poly1305
k = pqcrypto.randombytes(mac.klen)
m = pqcrypto.randombytes(1234567)
a = mac.auth(m,k)
mac.verify(a,m,k)
```

Beware that the key for a one-time authenticator must not be used for more than one message.

The following test script checks an AES-256-CTR ciphertext against Python's `Crypto` library:

```
import pqcrypto
cipher = pqcrypto.stream.aes256ctr

k = pqcrypto.randombytes(cipher.klen)
n = pqcrypto.randombytes(cipher.nlen)
m = pqcrypto.randombytes(1234567)
c = cipher.xor(m,n,k)
assert m == cipher.xor(c,n,k)

from Crypto.Cipher import AES
from Crypto.Util import Counter
import binascii

nint = int(binascii.hexlify(n),16)
s = AES.new(k,AES.MODE_CTR,counter=Counter.new(128,initial_value=nint))
assert s.encrypt(m) == c
```

Beware that nonces must be handled carefully in general, to avoid having the same nonce used for two messages under the same key; and even more carefully for AES-CTR, since each new 16-byte message block moves to a new nonce.

The following script has the same effect as the `pq-decrypt-mceliece8192128` command:

```
import os
import sys

import pqcrypto
kem = pqcrypto.kem.mceliece8192128
hash = pqcrypto.hash.shake256
enc = pqcrypto.stream.salsa20
auth = pqcrypto.onetimeauth.poly1305

with os.fdopen(0,"rb") as f: c = f.read()
with os.fdopen(8,"rb") as f: sk = f.read()
k = kem.dec(c[-kem.clen:],sk)
c = c[:-kem.clen]

h = hash(k)
kenc,h = h[:enc.klen],h[enc.klen:]
kauth = h[:auth.klen]

a,c = c[:auth.alen],c[auth.alen:]
auth.verify(a,c,kauth)

n = b"\0"*enc.nlen
m = enc.xor(c,n,kenc)
with os.fdopen(1,"wb") as f: f.write(m)
```

6 C library interface

The C API follows the principles of the NaCl/TweetNaCl/SUPERCOP/libsodium API, and in particular supports the previously defined `crypto_sign` and `crypto_kem` interfaces. However, to avoid namespace conflicts with NaCl, `libpqcrypto` uses `pqcrypto_*` names instead of `crypto_*` names. For example, put the following code into `testsign.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pqcrypto_sign_mqdss64.h"

unsigned char pk[pqcrypto_sign_mqdss64_PUBLICKEYBYTES];
unsigned char sk[pqcrypto_sign_mqdss64_SECRETKEYBYTES];

#define mlen 7
unsigned char m[mlen] = "hello\n";
unsigned char sm[pqcrypto_sign_mqdss64_BYTES + mlen];
unsigned long long smlen;
unsigned char t[sizeof sm];
unsigned long long tlen;

int main()
{
  if (pqcrypto_sign_mqdss64_keypair(pk,sk)) abort();
  if (pqcrypto_sign_mqdss64(sm,&smlen,m,mlen,sk)) abort();
  if (pqcrypto_sign_mqdss64_open(t,&tlen,sm,smlen,pk)) abort();
  if (tlen != mlen) abort();
  if (memcmp(t,m,mlen)) abort();
  return 0;
}
```

Compile and run as follows:

```
gcc -o testsign testsign.c \
  -I /home/libpqcrypto/include \
  -L /home/libpqcrypto/lib -Wl,-rpath=/home/libpqcrypto/lib \
  -lpqcrypto
./testsign && echo ok
```

The output will be `ok`.

If you have also compiled `x86` libraries on an `amd64` machine (see Section 3.3.5), you can compile in 32-bit mode as follows:

```
gcc -m32 -o testsign testsign.c \
  -I /home/libpqcrypto/include \
  -L /home/libpqcrypto/lib-x86 -Wl,-rpath=/home/libpqcrypto/lib-x86 \
  -lpqcrypto
```

libpqcrypto does not include NaCl-type selection of default primitives. The caller needs to select which pqcrypto_sign function to use and which pqcrypto_kem function to use.

Some of the software included in libpqcrypto uses malloc and is not suitable for environments that control memory usage statically.

7 Comparison to the implementations submitted to NIST

7.1 Namespacing

Each global symbol defined in libpqcrypto, and each header file provided by libpqcrypto, is in one of the following namespaces: pqcrypto; pqrandombytes; pqkernelrandombytes. The libpqcrypto compilation script issues warnings for any violations of this rule.

Implementations now follow this rule on several tested platforms (but this has not yet been comprehensively enforced at the source-code level and might still fail on other platforms). For externally visible functions, .c files now include crypto_kem.h, crypto_sign.h, randombytes.h, etc. For internal functions, new namespacing files use #define to move each internal function name into a private namespace, and are used via -imacros namespacing. Various .s files are now .S and also use the namespacing macros.

Some asm names were manually assigned in avx2 implementations in dilithium*, kyber*, newhope*, and ntruhrss*. These assignments are now gone in favor of namespacing.

Global symbols outside the defined API now have hidden visibility, preventing interposition when libpqcrypto is used as a shared library.

7.2 Randomness and tests

Various RNG software layers are now gone, including randombytes.c and randombytes.h in dilithium*; random.c in frodo*; prng_seed* in gui* and rainbow*; rand_bytes and randomness.h in picnic*; and various copies of NIST's rng.c and rng.h. All randomness is now obtained from randombytes(), provided by #include "randombytes.h". libpqcrypto includes a centralized randombytes() implementation, the same as fastrandombytes from SUPERCOP; and a centralized deterministic randombytes() implementation for checksums, the same as knownrandombytes from SUPERCOP. The deterministic implementation is used only for tests and is not included in -lpqcrypto.

The NIST KAT-generation code (PQCgen* and the simplified kat_*) is now gone. Other test drivers (test.c, test_qtesla.c, PQCtestKAT_sign.c, etc.) are also gone. The library has a centralized test/checksum mechanism, computed the same way as in SUPERCOP.

7.3 Compilation instructions

Each per-implementation Makefile is now gone. These files were used for several purposes:

- Specifying compiler choices (e.g., gcc -Ofast) and prerequisite libraries (e.g., -lkeccak -lcrypto). libpqcrypto handles this centrally.
- Specifying files to compile. libpqcrypto, like NaCl and SUPERCOP, always compiles all .c, .s, and .S files in the *top directory* of an implementation. Files in the aes and sha3 subdirectories for frodo*, and the sha3 subdirectory for qtesla*, are now in the top directory.

- Specifying files to *not* compile. libpqcrypto, like NaCl and SUPERCOP, does not compile files outside the top directory. Included files in the top directory under the name *.c that were not meant to be compiled directly (e.g., poly_mul.c in saber) are now renamed *.inc.

- For frodo*: Specifying various macros. Some of these were unused and are now eliminated. Others are now incorporated into .h files.

7.4 Bug fixes and portability improvements

dags* had a printf for the occasional "Non systematic matrix". This was caught by the automatic tests and is now removed.

frodo*/x64 had some vectorized _load_ and _store_ (rather than the safe _loadu_ and _storeu_) on data that was not necessarily aligned, crashing when the data was not aligned. This was caught by the automatic tests. The relevant arrays are now aligned.

gui* had a stack buffer overflow. This was caught by Address Sanitizer and is now fixed.

The luov*/portable implementation leaked memory. This was caught by valgrind and is now fixed.

mceliece*/avx used 0X for quad hex values in consts.S, and now uses 0x. 0X works with gcc but not clang.

mqdss64 was reading uninitialized data. This was caught by valgrind (and also indirectly by other tests) and is now fixed.

qtesla* allocated mlen bytes on the stack, crashing for messages above about 4 megabytes. This is now handled with malloc.

rainbow*a had a stack buffer overflow. This was caught by Address Sanitizer and is now fixed.

ramstake756839 wrote a zero byte past the end of the secret-key buffer. This was caught by the automatic tests and is now fixed.

sphincs* now includes various post-submission code updates.

7.5 Following existing interface rules

kindi* included a crypto_encrypt interface, but ignored the message length provided as input in that API, and instead assumed a fixed-length message. This was caught by the automatic tests. libpqcrypto provides only crypto_kem and crypto_sign, not crypto_encrypt.

dilithium*, gui*, luov*, mqdss*, qtesla*, rainbow*, and sphincs* did not allow the message pointer to match the signed-message pointer. This was caught by the automatic tests in some cases, including all cases where signatures were shorter than the message lengths in the tests. memcpy is now replaced by memmove where appropriate, and in some cases crypto_sign_open now copies the incoming signature to a temporary buffer.

dilithium*, gui*, and rainbow* did not allow the public-key pointer to match the output pointer in crypto_sign_open. This was caught by the automatic tests. crypto_sign_open now copies the public key to a temporary buffer.

7.6 Following additional interface rules

api.h is now stripped down to numeric definitions of CRYPTO_BYTES etc., so it can be easily parsed without C preprocessing.

The NIST submission rules were less restrictive and allowed `api.h` to be used as a general-purpose configuration mechanism. Various `.c` files that included `api.h` now include `apiorig.h` instead, with the original `api.h` renamed as `apiorig.h`. Probably some of these `apiorig.h` files can be removed, but this cleanup has not happened yet.

Each `kem` primitive now has a `goal-indcca2` file meaning that it tries to provide IND-CCA2 security (`libpqcrypto` does not include `newhope*cpa`), and a `goal-indcpa` file meaning that it tries to provide IND-CPA security (which is implied by IND-CCA2 security). However, the quantitative target security level is not indicated.

`mceliece*/avx`, `newhope*/avx2`, and `ntruhrss*/avx2` now use `rip`-relative addressing for constants in memory used in assembly. Previously they used absolute addressing, which works in a static library but not in a shared library.

Precomputed constants in `dilithium*`, `gui*`, `kyber*`, `luov*`, `mceliece*`, `newhope*`, `ntruhrss*`, `qtesla*`, `rainbow*`, and `ramstake*` are now defined as `const` (equivalently, `.section .rodata` in assembly) so that they are placed in the text segment and shared across processes. This is not comprehensive: some implementations of some primitives have variables in the data segment, and some functions (notably `randombytes`) are not thread-safe.

7.7 Fewer compiler warnings

By default, `libpqcrypto` compiles with `-Wall` with both `gcc` and `clang`. The following changes reduce the volume of warnings:

- `gui*`: The unused `num_nonzero_terms` function is now removed.
- `luov*` now closes comments in `parameters.h`. Various unused variables are now removed. An initializer `{0}` is replaced with `{{0}}` (and could simply be omitted).
- `qtesla*`: Various unused variables are now removed.
- `sphincs*` now has a revised `TRUNCSTORE` definition.
- `picnic*` now says `#ifndef api_h` instead of `#ifndef api_`.

7.8 More centralization

Most extracts from the Keccak Code Package (e.g., in `picnic*`) are now gone. The library has a centralized copy of a larger extract from the Keccak Code Package. However:

- `KeccakP-1600-times4-SIMD256.c` is still included in individual implementations since the Keccak Code Package does not have a portable implementation of the underlying `KeccakP1600times4_PermuteAll_24rounds` function.
- The Keccak Code Package is only one of the SHA-3 implementations; these implementations have not yet been merged.

`cpucycles()` implementations are now gone. The library has a centralized (and more portable) `cpucycles()`.

8 Notes on shared binaries

Most users install most packages through operating-system distributions such as Ubuntu. The packages are compiled and tested on a relatively small number of central systems, and are then installed as binaries on many more user systems. Similar comments apply to heterogeneous clusters of computers sharing binaries through NFS.

The idea of sharing binaries is limited by the fact that each binary runs efficiently on a limited set of CPUs: for example, the ARM CPU in a Raspberry Pi has a very different instruction set from an Intel or AMD CPU. If a binary is compiled for one CPU, and a user then tries running it on another CPU, then everything might be fine; or performance could be suboptimal but still acceptable; or performance could be so poor as to be unacceptable; or the binary could fail to run. There are "emulation" tools aiming at ensuring that all binaries run, but this still does not ensure acceptable performance.

Distributions deal with *part* of this problem by separately compiling binaries for different types of CPUs. For example, Debian distributes one set of binaries for 64-bit ARM CPUs (`arm64`), two sets for 32-bit ARM CPUs (`armhf` and `armel`), another set for 64-bit Intel/AMD CPUs (`amd64`), and another set for 32-bit Intel/AMD CPUs (`i386`). These are five of Debian's ten official "ports", and Debian also distributes many other unofficial "ports".

However, CPU manufacturers frequently release new "microarchitectures" that can run older binaries but that have new performance characteristics that allow new binaries to obtain better speeds. For example, Intel's "Sandy Bridge" microarchitecture (2011) added "AVX" support (some 256-bit vector instructions), and Intel's "Haswell" microarchitecture (2013) added "AVX2" support (more 256-bit vector instructions). For some important computations, including cryptographic computations, binaries that use AVX2 instructions will run much faster on Haswell (and newer) CPUs than binaries that do not; but if the binaries are distributed for *all* 64-bit Intel/AMD CPUs then they will be installed on some Sandy Bridge (and older) CPUs and will fail to run.

Operating-system distributions try to handle new microarchitectures in several ways, none of which are satisfactory:

- Prohibit use of new instructions such as AVX2. This works but produces a slowdown, often an unacceptable slowdown.
- Split off another "port" for the new processors. For example, `armhf` allows use of various instructions that `armel` does not allow. As a general rule, distributions resist the introduction of new ports, and new ports have historically been very slow to keep up with new microarchitectures. Each new port is a new hassle for users trying to figure out which port to choose, and a new hassle for distributions providing documentation to the users, even if all other aspects of porting are automated.
- Allow each package the option of also distributing *non-default* package variants that target the new processors. This is even more of a hassle for the users, who have no easy way to figure out which packages are best to install.

Without help from the distributions, programmers sometimes create "fat binaries" that include many different instruction sequences and that, at run time (when a program starts), inspect the CPU to predict which sequence will work best. However, reliably mapping various CPU details to the best instruction sequence is a software-engineering nightmare, limiting the deployment of this approach.

A much simpler, much more reliable way to select instruction sequences is to systematically benchmark each sequence, selecting the fastest sequence that works. However, doing this *at run time* is often unacceptably slow. An improvement is to perform systematic benchmarks at *install time* for performance-critical libraries (or at *boot time* to smoothly handle the occasional CPU replacements; a few functions are already benchmarked at boot time by the Linux kernel). This approach is taken in NaCl, and an improved version of this approach is taken in `libpqcrypto`. NaCl's benchmarking-and-installation process is tied to compilation; the improvement is that `libpqcrypto` has a first stage of compilation that does not need to be repeated (this can be performed centrally as part of preparing a "port"), and a second stage of selecting implementations.

9 Notes on building PKEs from KEMs

The command-line tools in `libpqcrypto` follow the traditional concept of public-key encryption systems (PKEs): they encrypt and decrypt user-specified messages.

The underlying encryption submissions to NIST instead specify key-encapsulation mechanisms (KEMs), which encapsulate and decapsulate random session keys. `libpqcrypto` converts each KEM to a PKE using the Cramer–Shoup "KEM-DEM" approach: the PKE uses the KEM to produce a session key, and then uses the session key as the key for an authenticated cipher that encrypts and authenticates the user's message.

`libpqcrypto` puts the DEM ciphertext *before* the KEM ciphertext. A receiver seeing the DEM ciphertext first is unable to decrypt it and must buffer it until seeing the KEM ciphertext. Common practice is to put the KEM ciphertext first, so that a receiver can immediately compute the session key and begin decrypting the DEM ciphertext without buffering; but this generally means releasing unverified plaintext, which is dangerous.[2] The order in `libpqcrypto` makes this dangerous behavior more difficult, although obviously still possible, to implement.

One way to attack a KEM-DEM is to guess the session key. If this attack is applied to T targets, each using a b-bit session key for the same plaintext, then each guess has probability approximately $T/2^b$ of success. For the typical choice $b = 256$, this attack is not a serious threat for any plausible value of T (and analogous quantum attacks are also not a serious threat). However, some KEMs use $b = 128$, and then the attack is a serious threat.

One can reduce $T/2^b$ to $1/2^b$ by deviating from the deterministic DEM framework: specifically, including a random number with the ciphertext and using this random number as a nonce for the authenticated cipher. However, it is simpler and stronger to skip this randomization and upgrade to KEMs that use $b \geq 256$. In the context of `libpqcrypto`, this means avoiding `frodokem640` (which uses $b = 128$) and `frodokem976` (which uses $b = 192$).

[2]A straightforward application of a PKE does not authenticate the sender, so the application receiving the data must be protected in some other way against forgeries. However, even when applications are not damaged by forgeries, they often give the attacker information about the plaintexts corresponding to some ciphertexts, including ciphertexts that the attacker obtained by modifying legitimate user ciphertexts whose plaintexts would *not* otherwise have been leaked. (Consider, e.g., a legitimate user plaintext that begins with a long user password known to the server and continues with a series of commands to be run under that user's credentials.) The authenticator in a DEM protects against this, but only if it is checked. "RUP security", as provided by some authenticated ciphers, does *not* protect against this; it only means that such leaks do not destroy the security of the cipher itself. A different solution is to split data into small packets, each packet being authenticated; this has a small bandwidth overhead, but has the advantage of not requiring buffering.

The authenticated cipher used in `libpqcrypto` follows a conventional encrypt-then-MAC framework, using a stream cipher to encrypt the user's message and a one-time authenticator to authenticate the ciphertext. `libpqcrypto` applies SHAKE256 to the session key to obtain the stream-cipher key and the authenticator key.

The stream cipher is Salsa20 (with nonce 0), and the one-time authenticator is Poly1305. The other choices mentioned in PQCRYPTO's initial recommendations in 2015 were AES-256 and GMAC, but those produce big slowdowns on small CPUs, with common speedup techniques leaking secrets through timing. AES also has quantitative security problems as a direct result of its small block size.

Of course, all details of this construction and implementation should be carefully audited.