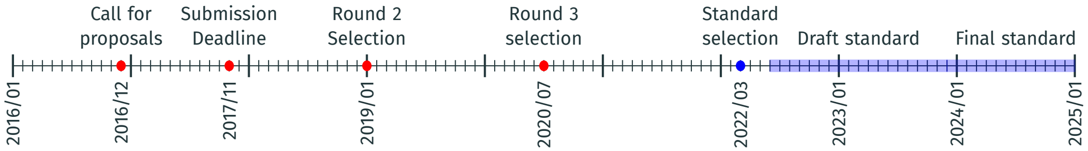
Improving Software Quality in Cryptography Standardization Projects

Matthias J. Kannwischer[1], Peter Schwabe[2,3], Douglas Stebila[4], and Thom Wiggers[3]

[1]Academia Sinica, Taiwan [2]MPI-SP, Germany
[3]Radboud University, The Netherlands
[4]University of Waterloo, Canada

06 June 2022, Security Standardisation Research Conference 2022

NIST Post-Quantum Competition



69 schemes 26 schemes 15 schemes 4? schemes

- NIST will announce their selection in March 2022
- No announcement until today (98th March 2022)

Motivation

"NISTPQC, despite being an important and timely project, has produced the largest regression ever in the quality of cryptographic software. This will not be easy to fix."

– Daniel J. Bernstein, 2018

Motivation

- We are authors + maintainers of
  - **liboqs** (prototyping PQC in TLS/SSH): `https://github.com/open-quantum-safe/liboqs`
  - **pqm4** (PQC for microcontrollers): `https://github.com/mupq/pqm4`
- Authored many PQC implementation papers
- Starting point is always the reference code of the NISTPQC submissions
- Software quality of many initial submissions was very bad
- It got significantly better in later rounds
  - Schemes with particularly bad software got eliminated by NIST
  - Submission teams improved their implementations
  - A lot of feedback provided (by us/SUPERCOP/many others) to the submission teams

Motivation

- We are authors + maintainers of
  - **liboqs** (prototyping PQC in TLS/SSH): `https://github.com/open-quantum-safe/liboqs`
  - **pqm4** (PQC for microcontrollers): `https://github.com/mupq/pqm4`
- Authored many PQC implementation papers
- Starting point is always the reference code of the NISTPQC submissions
- Software quality of many initial submissions was very bad
- It got significantly better in later rounds
  - Schemes with particularly bad software got eliminated by NIST
  - Submission teams improved their implementations
  - A lot of feedback provided (by us/SUPERCOP/many others) to the submission teams

Motivation

- We are authors + maintainers of
  - **liboqs** (prototyping PQC in TLS/SSH): `https://github.com/open-quantum-safe/liboqs`
  - **pqm4** (PQC for microcontrollers): `https://github.com/mupq/pqm4`
- Authored many PQC implementation papers
- Starting point is always the reference code of the NISTPQC submissions
- Software quality of many initial submissions was very bad
- It got significantly better in later rounds
  - Schemes with particularly bad software got eliminated by NIST
  - Submission teams improved their implementations
  - A lot of feedback provided (by us/SUPERCOP/many others) to the submission teams

Motivation

- We are authors + maintainers of
  - **liboqs** (prototyping PQC in TLS/SSH): `https://github.com/open-quantum-safe/liboqs`
  - **pqm4** (PQC for microcontrollers): `https://github.com/mupq/pqm4`
- Authored many PQC implementation papers
- Starting point is always the reference code of the NISTPQC submissions
- Software quality of many initial submissions was very bad
- It got significantly better in later rounds
  - Schemes with particularly bad software got eliminated by NIST
  - Submission teams improved their implementations
  - A lot of feedback provided (by us/SUPERCOP/many others) to the submission teams

Motivation

- Motivation for **PQClean**: Clean up the code once
    - Active project since January 2018
    - Joint work with Rijneveld, Schanck, Schwabe, Stebila, Wiggers
    - Integration into liboqs/pqm4
    - Used in our own research
    - Used by other researchers
- Motivation for **this paper**
    - NISTPQC is nearing its end
    - More competitions to come (e.g., NIST will call for more PQC signatures soon)
    - Describe lessons learned from PQClean
    - **What could NIST do better in the next competition?**
    - We think: Properly defined software quality guidelines together with a testing framework can save everyone (including NIST) a lot of time

Motivation

- Motivation for **PQClean**: Clean up the code once
  - Active project since January 2018
  - Joint work with Rijneveld, Schanck, Schwabe, Stebila, Wiggers
  - Integration into liboqs/pqm4
  - Used in our own research
  - Used by other researchers
- Motivation for **this paper**
  - NISTPQC is nearing its end
  - More competitions to come (e.g., NIST will call for more PQC signatures soon)
  - Describe lessons learned from PQClean
  - **What could NIST do better in the next competition?**
  - We think: Properly defined software quality guidelines together with a testing framework can save everyone (including NIST) a lot of time

Motivation

- Motivation for **PQClean**: Clean up the code once
  - Active project since January 2018
  - Joint work with Rijneveld, Schanck, Schwabe, Stebila, Wiggers
  - Integration into liboqs/pqm4
  - Used in our own research
  - Used by other researchers
- Motivation for **this paper**
  - NISTPQC is nearing its end
  - More competitions to come (e.g., NIST will call for more PQC signatures soon)
  - Describe lessons learned from PQClean
  - **What could NIST do better in the next competition?**
  - We think: Properly defined software quality guidelines together with a testing framework can save everyone (including NIST) a lot of time

Motivation

- Motivation for **PQClean**: Clean up the code once
    - Active project since January 2018
    - Joint work with Rijneveld, Schanck, Schwabe, Stebila, Wiggers
    - Integration into liboqs/pqm4
    - Used in our own research
    - Used by other researchers
- Motivation for **this paper**
    - NISTPQC is nearing its end
    - More competitions to come (e.g., NIST will call for more PQC signatures soon)
    - Describe lessons learned from PQClean
    - **What could NIST do better in the next competition?**
    - We think: Properly defined software quality guidelines together with a testing framework can save everyone (including NIST) a lot of time

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to “promote understanding of how the submitted algorithm may be implemented”, in which “clarity… is more important than… efficiency” (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations’ owners granting certain rights to use the implementation “for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard” (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to “promote understanding of how the submitted algorithm may be implemented”, in which “clarity… is more important than… efficiency” (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations’ owners granting certain rights to use the implementation “for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard” (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to "promote understanding of how the submitted algorithm may be implemented", in which "clarity… is more important than… efficiency" (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations' owners granting certain rights to use the implementation "for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard" (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to “promote understanding of how the submitted algorithm may be implemented”, in which “clarity... is more important than... efficiency” (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations’ owners granting certain rights to use the implementation “for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard” (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to "promote understanding of how the submitted algorithm may be implemented", in which "clarity… is more important than… efficiency" (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations' owners granting certain rights to use the implementation "for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard" (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Submission requirements (December 2016 – link)

- A reference implementation, written in ANSI C, intended to “promote understanding of how the submitted algorithm may be implemented”, in which “clarity... is more important than... efficiency” (§2.C.1)
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor
- A statement by the implementations’ owners granting certain rights to use the implementation “for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard” (§2.D.3)
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations (§5.B).
- NIST provided a document describing the required APIs and code to generate the KAT values.

NIST requirements

Evaluation criteria (December 2016 – link):

- Performance: schemes will be evaluated based on their computational cost in software and hardware (§4.B.2).
- Side channel aspects: schemes that can be made side-channel resistant efficiently are more desirable than those that cannot, and “optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not” (§4.A.6).
- Flexibility: schemes that “can be implemented securely and efficiently on a wide variety of platforms” or for which implementations “can be parallelized to achieve higher performance” are desirable (§4.C.1).

NIST requirements

Evaluation criteria (December 2016 – link):

- Performance: schemes will be evaluated based on their computational cost in software and hardware (§4.B.2).
- Side channel aspects: schemes that can be made side-channel resistant efficiently are more desirable than those that cannot, and “optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not” (§4.A.6).
- Flexibility: schemes that “can be implemented securely and efficiently on a wide variety of platforms” or for which implementations “can be parallelized to achieve higher performance” are desirable (§4.C.1).

NIST requirements

Evaluation criteria (December 2016 – link):

- Performance: schemes will be evaluated based on their computational cost in software and hardware (§4.B.2).
- Side channel aspects: schemes that can be made side-channel resistant efficiently are more desirable than those that cannot, and “optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not” (§4.A.6).
- Flexibility: schemes that “can be implemented securely and efficiently on a wide variety of platforms” or for which implementations “can be parallelized to achieve higher performance” are desirable (§4.C.1).

Bad Crypto Software: Examples

- Let’s look at some examples of bad code in the NISTPQC competition
- This is not to point fingers at submission teams
  - Cryptographers are not software engineers
  - Our opinion: Organizer of crypto competition is to blame for many problems

Bad Crypto Software: Examples

- Let’s look at some examples of bad code in the NISTPQC competition
- This is not to point fingers at submission teams
  - Cryptographers are not software engineers
  - Our opinion: Organizer of crypto competition is to blame for many problems

Bad Crypto Software: Examples

- Let’s look at some examples of bad code in the NISTPQC competition
- This is not to point fingers at submission teams
  - Cryptographers are not software engineers
  - Our opinion: Organizer of crypto competition is to blame for many problems

Bad NISTPQC Software: Example 1

```
int64_t* extEuclid(int64_t a, int64_t b) {
    int64_t array[3];
    int64_t *dxy = array;
    ...
    return dxy;
}
```

- Returns pointer to local stack $\rightarrow$ Undefined behavior
- Luckily: Function was never used $\rightarrow$ Should still eliminate this dead code

Bad NISTPQC Software: Example 1

```
int64_t* extEuclid(int64_t a, int64_t b) {
    int64_t array[3];
    int64_t *dxy = array;
    ...
    return dxy;
}
```

- Returns pointer to local stack $\rightarrow$ Undefined behavior
- Luckily: Function was never used $\rightarrow$ Should still eliminate this dead code

Bad NISTPQC Software: Example 2

```
#define CRYPTO_BYTES XXX               // size of the shared secret
#define CRYPTO_CIPHERTEXTBYTES XXX     // size of the ciphertext
#define CRYPTO_PUBLICKEYBYTES XXX      // size of the public key
#define CRYPTO_SECRETKEYBYTES XXX      // size of the secret key

...
int crypto_kem_dec(unsigned char k[CRYPTO_BYTES],
                  const unsigned char ct[CRYPTO_CIPHERTEXTBYTES],
                  const unsigned char sk[CRYPTO_SECRETKEYBYTES]) {
...
    unsigned char x = sk[CRYPTO_SECRETKEYBYTES + j];
...
}
```

Bad NISTPQC Software: Example 2

- What?
- Obvious read out of bounds
- Recall: Most PQC KEMs use the Fujisaki–Okamoto (for transforming CPA to CCA)
    - Requires to re-encrypt the decrypted message, i.e., requires public key in decapsulation
    - Correct solution: Store a copy of the public key in the secret key
    - Here: Just assume pk is in memory behind sk
      $\rightarrow$ Tests work just fine
      $\rightarrow$ Breaks for any real application / when compiler decides to put pk in different place

Bad NISTPQC Software: Example 2

- What?
- Obvious read out of bounds
- Recall: Most PQC KEMs use the Fujisaki–Okamoto (for transforming CPA to CCA)
  - Requires to re-encrypt the decrypted message, i.e., requires public key in decapsulation
  - Correct solution: Store a copy of the public key in the secret key
  - Here: Just assume pk is in memory behind sk
    $\to$ Tests work just fine
    $\to$ Breaks for any real application / when compiler decides to put pk in different place

Bad NISTPQC Software: Example 2

- What?
- Obvious read out of bounds
- Recall: Most PQC KEMs use the Fujisaki–Okamoto (for transforming CPA to CCA)
    - Requires to re-encrypt the decrypted message, i.e., requires public key in decapsulation
    - Correct solution: Store a copy of the public key in the secret key
    - Here: Just assume pk is in memory behind sk
      $\rightarrow$ Tests work just fine
      $\rightarrow$ Breaks for any real application / when compiler decides to put pk in different place

Bad NISTPQC Software: Example 2

- What?
- Obvious read out of bounds
- Recall: Most PQC KEMs use the Fujisaki–Okamoto (for transforming CPA to CCA)
    - Requires to re-encrypt the decrypted message, i.e., requires public key in decapsulation
    - Correct solution: Store a copy of the public key in the secret key
    - Here: Just assume pk is in memory behind sk
      $\rightarrow$ Tests work just fine
      $\rightarrow$ Breaks for any real application / when compiler decides to put pk in different place

Bad NISTPQC Software: Example 2

- What?
- Obvious read out of bounds
- Recall: Most PQC KEMs use the Fujisaki–Okamoto (for transforming CPA to CCA)
  - Requires to re-encrypt the decrypted message, i.e., requires public key in decapsulation
  - Correct solution: Store a copy of the public key in the secret key
  - Here: Just assume pk is in memory behind sk
    $\rightarrow$ Tests work just fine
    $\rightarrow$ Breaks for any real application / when compiler decides to put pk in different place

Bad NISTPQC Software: Example 3

```
...
uint32_t t; uint8_t buf[BUFLEN]; uint32_t a[N];
...
// rejection sample a[i] <= THRESHOLD  (20 bits = 2.5 bytes per coeff)
while(ctr < N) {
    t  = buf[pos]; t |= (uint32_t)buf[pos + 1] << 8;
    t |= (uint32_t)buf[pos + 2] << 16; t &= 0xFFFFF;

    t  = buf[pos + 2] >> 4; t |= (uint32_t)buf[pos + 3] << 4;
    t |= (uint32_t)buf[pos + 4] << 12; pos += 5;

    if(t <= THRESHOLD) a[ctr++] = t;
    if(t <= THRESHOLD) a[ctr++] = t;

    ...
}
...
```

Bad NISTPQC Software: Example 3

- Code works perfectly fine
- Subtle bug: `t` is overwritten before it is used
  $\rightarrow$ $a[i]$ and $a[i+1]$ are the same
  $\rightarrow$ Here: Bad enough to completely leak the secret key
- Fix: Use two variables `t0` and `t1`

Bad NISTPQC Software: Example 3

- Code works perfectly fine
- Subtle bug: `t` is overwritten before it is used
  $\rightarrow$ $a[i]$ and $a[i+1]$ are the same
  $\rightarrow$ Here: Bad enough to completely leak the secret key
- Fix: Use two variables `t0` and `t1`

Bad NISTPQC Software: Example 3

- Code works perfectly fine
- Subtle bug: `t` is overwritten before it is used
  $\rightarrow$ $a[i]$ and $a[i+1]$ are the same
  $\rightarrow$ Here: Bad enough to completely leak the secret key
- Fix: Use two variables `t0` and `t1`

Bad NISTPQC Software: Other problems

- **Alignment assumptions**: You cannot assume that `uint8_t *ptr` is aligned
  Upcast is undefined behavior: `(uint32_t *)ptr` or `(__m256i *)ptr`
- **Timing leaks** in many implementations
  (Many implementations did not claim to be constant-time though)
- **Unclear licensing**
  Many submissions did not provide a LICENSE file
  (For one submission we did not get one even after asking for many years)

Bad NISTPQC Software: Other problems

- **Alignment assumptions**: You cannot assume that `uint8_t *ptr` is aligned
  Upcast is undefined behavior: `(uint32_t *)ptr` or `(__m256i *)ptr`
- **Timing leaks** in many implementations
  (Many implementations did not claim to be constant-time though)
- **Unclear licensing**
  Many submissions did not provide a LICENSE file
  (For one submission we did not get one even after asking for many years)

Bad NISTPQC Software: Other problems

- **Alignment assumptions**: You cannot assume that `uint8_t *ptr` is aligned
  Upcast is undefined behavior: `(uint32_t *)ptr` or `(__m256i *)ptr`
- **Timing leaks** in many implementations
  (Many implementations did not claim to be constant-time though)
- **Unclear licensing**
  Many submissions did not provide a LICENSE file
  (For one submission we did not get one even after asking for many years)

How could this be solved?

- Cryptographers answer: **High-assurance cryptography**
    - Formally verify all the code
    - Ultimate goal for standards before they are put into production
    - ... this takes years
- However, many bugs can be caught by **standard techniques from software development**
    - Why not use them?
    - Goal: Save us time in evaluation; not guarantee correctness

How could this be solved?

- Cryptographers answer: **High-assurance cryptography**
  - Formally verify all the code
  - Ultimate goal for standards before they are put into production
  - ... this takes years
- However, many bugs can be caught by **standard techniques from software development**
  - Why not use them?
  - Goal: Save us time in evaluation; not guarantee correctness

How could this be solved?

- Cryptographers answer: **High-assurance cryptography**
    - Formally verify all the code
    - Ultimate goal for standards before they are put into production
    - ... this takes years
- However, many bugs can be caught by **standard techniques from software development**
    - Why not use them?
    - Goal: Save us time in evaluation; not guarantee correctness

How could this be solved?

- Cryptographers answer: **High-assurance cryptography**
  - Formally verify all the code
  - Ultimate goal for standards before they are put into production
  - ... this takes years
- However, many bugs can be caught by **standard techniques from software development**
  - Why not use them?
  - Goal: Save us time in evaluation; not guarantee correctness

Our solution

```
-Wall -Werror
-Wpedantic -Wextra
```

Our solution: PQClean – Features

GitHub repo with extensive CI to ensure “clean” implementations:
`https://github.com/PQClean/PQClean`

- Goal: collect “clean C” code of all NISTPQC schemes
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization
- Later
  - Also integrate optimized implementations (avx2, aarch64)
  - Implementations in other languages (?)

Our solution: PQClean – Features

GitHub repo with extensive CI to ensure “clean” implementations:
`https://github.com/PQClean/PQClean`

- Goal: collect “clean C” code of all NISTPQC schemes
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization
- Later
  - Also integrate optimized implementations (avx2, aarch64)
  - Implementations in other languages (?)

Our solution: PQClean – Features

GitHub repo with extensive CI to ensure “clean” implementations:
`https://github.com/PQClean/PQClean`

- Goal: collect “clean C” code of all NISTPQC schemes
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization
- Later
  - Also integrate optimized implementations (avx2, aarch64)
  - Implementations in other languages (?)

Our solution: PQClean – Features

GitHub repo with extensive CI to ensure “clean” implementations:
`https://github.com/PQClean/PQClean`

- Goal: collect “clean C” code of all NISTPQC schemes
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization
- Later
  - Also integrate optimized implementations (avx2, aarch64)
  - Implementations in other languages (?)

Our solution: PQClean – Features

What we consider clean

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- API functions do not need pointers to be aligned
- Builds under Linux, MacOS, and Windows
- Compiles with `-Wall -Wextra -Wpedantic -Werror` with gcc and clang
- Compiles with `/W4 /WX` with MS compiler
- Consistent test vectors across runs
- Consistent test vectors on big-endian and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines

Our solution: PQClean – Features

What we consider clean

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- API functions do not need pointers to be aligned
- Builds under Linux, MacOS, and Windows
- Compiles with `-Wall -Wextra -Wpedantic -Werror` with gcc and clang
- Compiles with `/W4 /WX` with MS compiler
- Consistent test vectors across runs
- Consistent test vectors on big-endian and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines

Our solution: PQClean – Features

What we consider clean

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- API functions do not need pointers to be aligned
- Builds under Linux, MacOS, and Windows
- Compiles with `-Wall -Wextra -Wpedantic -Werror` with gcc and clang
- Compiles with `/W4 /WX` with MS compiler
- Consistent test vectors across runs
- Consistent test vectors on big-endian and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines

Our solution: PQClean – Features

What we consider clean

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- API functions do not need pointers to be aligned
- Builds under Linux, MacOS, and Windows
- Compiles with `-Wall -Wextra -Wpedantic -Werror` with gcc and clang
- Compiles with `/W4 /WX` with MS compiler
- Consistent test vectors across runs
- Consistent test vectors on big-endian and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines

Our solution: PQClean – Features

What we consider clean

- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- No errors/warnings reported by undefined-behavior sanitizer
- No dynamic memory allocations
- No external dependencies (except for AES/SHA-2/SHA-3)
- Each implementation comes with license and meta information in `META.yml`
- No variable-length arrays (required to build under Windows)

Our solution: PQClean – Features

What we consider clean

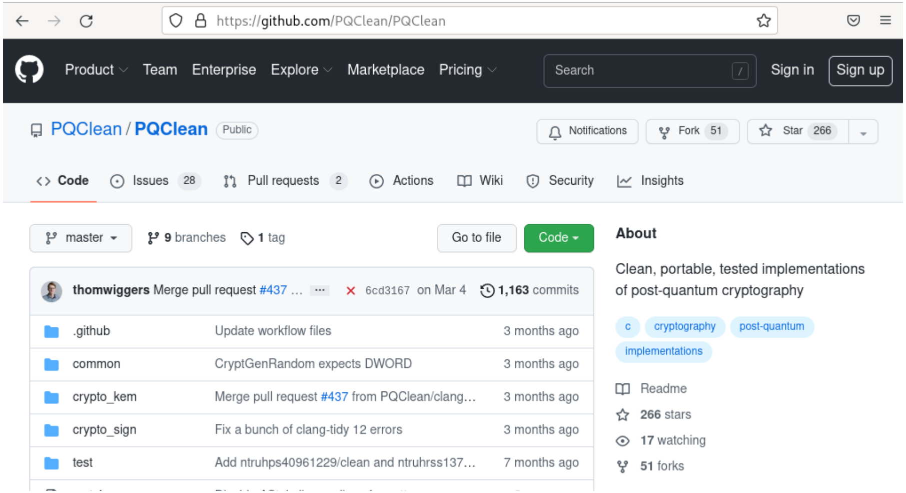
- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- No errors/warnings reported by undefined-behavior sanitizer
- No dynamic memory allocations
- No external dependencies (except for AES/SHA-2/SHA-3)
- Each implementation comes with license and meta information in `META.yml`
- No variable-length arrays (required to build under Windows)

Our solution: PQClean – Features

What we consider clean

- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- No errors/warnings reported by undefined-behavior sanitizer
- No dynamic memory allocations
- No external dependencies (except for AES/SHA-2/SHA-3)
- Each implementation comes with license and meta information in `META.yml`
- No variable-length arrays (required to build under Windows)

Our solution: PQClean



https://github.com/PQClean/PQClean

PQClean / PQClean Public

thomwiggers Merge pull request #437 … 6cd3167 on Mar 4 1,163 commits

.github	Update workflow files	3 months ago
common	CryptGenRandom expects DWORD	3 months ago
crypto_kem	Merge pull request #437 from PQClean/clang…	3 months ago
crypto_sign	Fix a bunch of clang-tidy 12 errors	3 months ago
test	Add ntruhps40961229/clean and ntruhrss137…	7 months ago

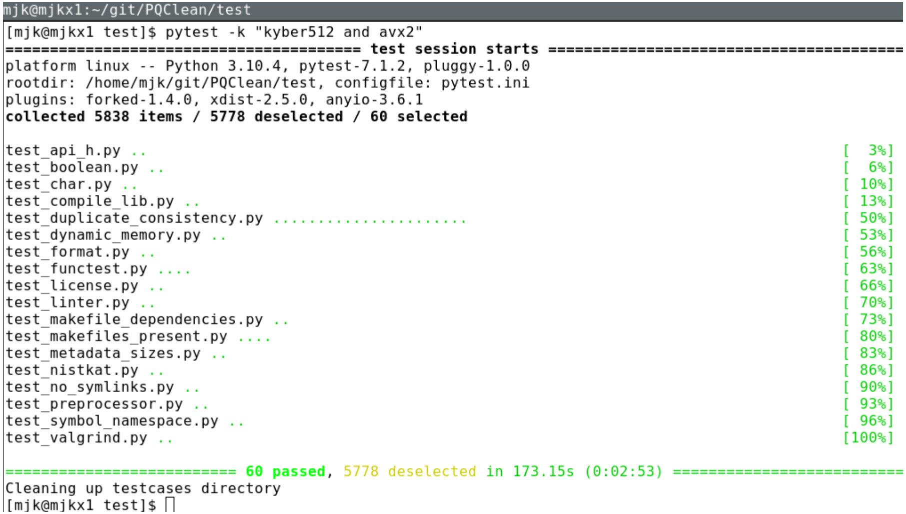
About

Clean, portable, tested implementations of post-quantum cryptography

c cryptography post-quantum implementations

Readme
266 stars
17 watching
51 forks

Our solution: PQClean – Running tests locally

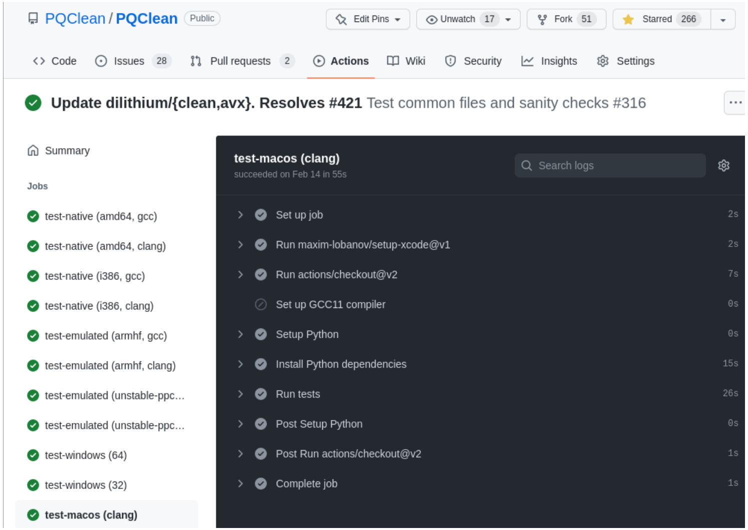


```
mjk@mjkx1:~/git/PQClean/test
[mjk@mjkx1 test]$ pytest -k "kyber512 and avx2"
======================================== test session starts ========================================
platform linux -- Python 3.10.4, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/mjk/git/PQClean/test, configfile: pytest.ini
plugins: forked-1.4.0, xdist-2.5.0, anyio-3.6.1
collected 5838 items / 5778 deselected / 60 selected

test_api_h.py ..                                                                              [  3%]
test_boolean.py ..                                                                            [  6%]
test_char.py ..                                                                               [ 10%]
test_compile_lib.py ..                                                                        [ 13%]
test_duplicate_consistency.py ......................                                          [ 50%]
test_dynamic_memory.py ..                                                                     [ 53%]
test_format.py ..                                                                             [ 56%]
test_functest.py ....                                                                         [ 63%]
test_license.py ..                                                                            [ 66%]
test_linter.py ..                                                                             [ 70%]
test_makefile_dependencies.py ..                                                              [ 73%]
test_makefiles_present.py ....                                                                [ 80%]
test_metadata_sizes.py ..                                                                     [ 83%]
test_nistkat.py ..                                                                            [ 86%]
test_no_symlinks.py ..                                                                        [ 90%]
test_preprocessor.py ..                                                                       [ 93%]
test_symbol_namespace.py ..                                                                   [ 96%]
test_valgrind.py ..                                                                           [100%]

========================== 60 passed, 5778 deselected in 173.15s (0:02:53) ==========================
Cleaning up testcases directory
[mjk@mjkx1 test]$
```

Our solution: PQClean – Tests run automatically by CI for every pull request



PQClean / PQClean Public

Edit Pins · Unwatch 17 · Fork 51 · Starred 266

Code · Issues 28 · Pull requests 2 · Actions · Wiki · Security · Insights · Settings

✓ **Update dilithium/{clean,avx}. Resolves #421** Test common files and sanity checks #316

Summary

Jobs

- ✓ test-native (amd64, gcc)
- ✓ test-native (amd64, clang)
- ✓ test-native (i386, gcc)
- ✓ test-native (i386, clang)
- ✓ test-emulated (armhf, gcc)
- ✓ test-emulated (armhf, clang)
- ✓ test-emulated (unstable-ppc…
- ✓ test-emulated (unstable-ppc…
- ✓ test-windows (64)
- ✓ test-windows (32)
- ✓ **test-macos (clang)**

test-macos (clang)
succeeded on Feb 14 in 55s

Search logs

```
Set up job                          2s
Run maxim-lobanov/setup-xcode@v1    2s
Run actions/checkout@v2             7s
Set up GCC11 compiler               0s
Setup Python                        0s
Install Python dependencies        15s
Run tests                          26s
Post Setup Python                   0s
Post Run actions/checkout@v2        1s
Complete job                        1s
```

Findings from PQClean

TABLE 1. FLAWS FOUND, AND WHICH TESTS MIGHT HAVE DETECTED THEM, IN HOW MANY OF THE 10 KEMS AND 7 SIGNATURE SCHEMES THAT HAVE EVER BEEN INCLUDED IN PQCLEAN.

Flaw	KEMs	Sigs	**Flaw**	KEMs	Sigs		**Test**
Memory safety ⋄	3	4	Endianness assumptions †	7	2	⋆	Compilation test
Signed integer overflow ⋆, ⋄, ✝	3	1	Platform-specific behavior ♣, ±, †, ✝	4	0	♠	`Makefile` checks
Alignment assumptions ⋆, ♣, ⋄	4	4	Variable-Length Arrays ⋆	4	1	♣	Functional tests
Other Undefined Behavior ⋆, ♣	1	1	Compiler extensions ⋆	5	2	†	Test vectors
Dead code ⋆, ♠	3	4	Integer sizes ⋄, ⋆, †	6	3	⋄	Sanitizers
Global state	2	1	Non-constant time =	4	0	±	Signedness of `char`
Licensing unclear ©	3	1				=	Timing-suspicious ops.
						✝	`clang-tidy`
						©	License file

Proposal for the next competition

- We think it is up to the standardizing entity to define guidelines and a framework
- Make some guidelines mandatory for the first submission and others optional
- After submission: Publish list of warnings/errors and give submitters time to fix it
- Later round: All mandatory

Proposal for the next competition

- We think it is up to the standardizing entity to define guidelines and a framework
- Make some guidelines mandatory for the first submission and others optional
- After submission: Publish list of warnings/errors and give submitters time to fix it
- Later round: All mandatory

Proposal for the next competition

- We think it is up to the standardizing entity to define guidelines and a framework
- Make some guidelines mandatory for the first submission and others optional
- After submission: Publish list of warnings/errors and give submitters time to fix it
- Later round: All mandatory

Proposal for the next competition – Recommendations

- At least 6 months before the submission deadline:
  - Provide a build system with a reasonable level of compiler warnings turned on
    - Basic: Offline build system (w/ virtualization)
    - Ideally: Continuous integration setup in the Cloud – Github Actions looks promising
  - Provide a working example (e.g., pre-quantum RSA KEM and signature)
  - Provide common building blocks (AES, SHA-2, SHA-3)
  - Automated functional testing
  - Verify testvectors automatically
  - Automated testing using all major toolchains (gcc, clang, MS compiler)
  - Automated testing on all major platform (x86, aarch64)
  - Use modern static and dynamic analysis (Valgrind, asan, ...)

Proposal for the next competition – Recommendations

- At least 6 months before the submission deadline:
  - Provide a build system with a reasonable level of compiler warnings turned on
    - Basic: Offline build system (w/ virtualization)
    - Ideally: Continuous integration setup in the Cloud – Github Actions looks promising
  - Provide a working example (e.g., pre-quantum RSA KEM and signature)
  - Provide common building blocks (AES, SHA-2, SHA-3)
  - Automated functional testing
  - Verify testvectors automatically
  - Automated testing using all major toolchains (gcc, clang, MS compiler)
  - Automated testing on all major platform (x86, aarch64)
  - Use modern static and dynamic analysis (Valgrind, asan, ...)

Proposal for the next competition – Recommendations

- At least 6 months before the submission deadline:
  - Provide a build system with a reasonable level of compiler warnings turned on
    - Basic: Offline build system (w/ virtualization)
    - Ideally: Continuous integration setup in the Cloud – Github Actions looks promising
  - Provide a working example (e.g., pre-quantum RSA KEM and signature)
  - Provide common building blocks (AES, SHA-2, SHA-3)
  - Automated functional testing
  - Verify testvectors automatically
  - Automated testing using all major toolchains (gcc, clang, MS compiler)
  - Automated testing on all major platform (x86, aarch64)
  - Use modern static and dynamic analysis (Valgrind, asan, ...)

Proposal for the next competition – Recommendations

- At least 6 months before the submission deadline:
  - Provide a build system with a reasonable level of compiler warnings turned on
    - Basic: Offline build system (w/ virtualization)
    - Ideally: Continuous integration setup in the Cloud – Github Actions looks promising
  - Provide a working example (e.g., pre-quantum RSA KEM and signature)
  - Provide common building blocks (AES, SHA-2, SHA-3)
  - Automated functional testing
  - Verify testvectors automatically
  - Automated testing using all major toolchains (gcc, clang, MS compiler)
  - Automated testing on all major platform (x86, aarch64)
  - Use modern static and dynamic analysis (Valgrind, asan, ...)

Proposal for the next competition – Recommendations

- At least 6 months before the submission deadline:
  - Provide a build system with a reasonable level of compiler warnings turned on
    - Basic: Offline build system (w/ virtualization)
    - Ideally: Continuous integration setup in the Cloud – Github Actions looks promising
  - Provide a working example (e.g., pre-quantum RSA KEM and signature)
  - Provide common building blocks (AES, SHA-2, SHA-3)
  - Automated functional testing
  - Verify testvectors automatically
  - Automated testing using all major toolchains (gcc, clang, MS compiler)
  - Automated testing on all major platform (x86, aarch64)
  - Use modern static and dynamic analysis (Valgrind, asan, ...)

Proposal for the next competition – Advanced

- Enforce namespacing
- Enforce code style and documentation
- Benchmarking (?)
- Verify that code is constant-time
- Disallow dynamic memory allocations (?) – embedded implementations vs. stack size limits

Proposal for the next competition – Advanced

- Enforce namespacing
- Enforce code style and documentation
- Benchmarking (?)
- Verify that code is constant-time
- Disallow dynamic memory allocations (?) – embedded implementations vs. stack size limits

Proposal for the next competition – Advanced

- Enforce namespacing
- Enforce code style and documentation
- Benchmarking (?)
- Verify that code is constant-time
- Disallow dynamic memory allocations (?) – embedded implementations vs. stack size limits

Proposal for the next competition – Advanced

- Enforce namespacing
- Enforce code style and documentation
- Benchmarking (?)
- Verify that code is constant-time
- Disallow dynamic memory allocations (?) – embedded implementations vs. stack size limits

Proposal for the next competition – Advanced

- Enforce namespacing
- Enforce code style and documentation
- Benchmarking (?)
- Verify that code is constant-time
- Disallow dynamic memory allocations (?) – embedded implementations vs. stack size limits

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
  - Spend much more time on software guidelines and a testing framework
  - Make use of what we learned from PQClean (it's not perfect)
  - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over
  – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
  - Spend much more time on software guidelines and a testing framework
  - Make use of what we learned from PQClean (it's not perfect)
  - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
    - Spend much more time on software guidelines and a testing framework
    - Make use of what we learned from PQClean (it's not perfect)
    - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
  - Spend much more time on software guidelines and a testing framework
  - Make use of what we learned from PQClean (it's not perfect)
  - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
  - Spend much more time on software guidelines and a testing framework
  - Make use of what we learned from PQClean (it's not perfect)
  - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions

- NISTPQC was a disaster in terms of software quality (at least in the first round)
- Researchers wasted many hours on fixing the same bugs over and over – this has cost many PhD student years
- PQClean helped to catch a lot of bugs
- Other projects like SUPERCOP were also very useful
- Recommendations for NIST
  - Spend much more time on software guidelines and a testing framework
  - Make use of what we learned from PQClean (it's not perfect)
  - A basic framework is much better than none
- Recommendation for everyone: Please, `-Wall -Werror -Wpedantic -Wextra`

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
  - “Executable pseudo-code”
  - More and more common
  - Not suitable for optimized code
- **Rust**
  - Solves many of the memory problems of C
  - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
    - “Executable pseudo-code”
    - More and more common
    - Not suitable for optimized code
- **Rust**
    - Solves many of the memory problems of C
    - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
  - “Executable pseudo-code”
  - More and more common
  - Not suitable for optimized code
- **Rust**
  - Solves many of the memory problems of C
  - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
    - “Executable pseudo-code”
    - More and more common
    - Not suitable for optimized code
- **Rust**
    - Solves many of the memory problems of C
    - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
    - “Executable pseudo-code”
    - More and more common
    - Not suitable for optimized code
- **Rust**
    - Solves many of the memory problems of C
    - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Conclusions – Beyond C

- Is C a good language for reference implementations? – No
- **Python**
    - “Executable pseudo-code”
    - More and more common
    - Not suitable for optimized code
- **Rust**
    - Solves many of the memory problems of C
    - Suitable for optimized code
- **SageMath**/**Magma**: Abstracts away many of the mathematical constructions
- **hacspec**: Support formal verification

- No consensus on what is best suited
- Community and standardization entities should re-evaluate this regularly

Thank you very much for your attention!

matthias@kannwischer.eu

Paper: `https://eprint.iacr.org/2022/337`

Code: `https://github.com/PQClean/PQClean`