pqm4: Benchmarking PQC on the Cortex-M4

Matthias J. Kannwischer
Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu

May 02, 2019, NIST PQC Hardware Day, NIST, Gaithersburg
IS THIS HARDWARE?
pqm4: Benchmarking PQC on the Cortex-M4

Matthias J. Kannwischer
Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu

May 02, 2019, NIST PQC Hardware Embedded Platforms Day, NIST, Gaithersburg
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first

As recommended by NIST

Majority of cryptographic devices is way smaller

- Limited RAM
- No/limited vector instructions
- Side-channels?

"Performance will play a larger role in the second round"

Some questions:

- What reference platform should we use?
- Are schemes efficient on small ARMs?
- What are the requirements for RAM and code-size?
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller

Some questions:
- What reference platform should we use?
- Are schemes efficient on small ARMs?
- What are the requirements for RAM and code-size?
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM

Some questions:
- What reference platform should we use?
- Are schemes efficient on small ARMs?
- What are the requirements for RAM and code-size?
Motivation

- Submission teams focused on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
Motivation

- Submission teams focused on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
- “Performance will play a larger role in the second round”
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
- “Performance will play a larger role in the second round”
- Some questions:
Motivation

- Submission teams focused on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
- "Performance will play a larger role in the second round"
- Some questions:
  - What reference platform should we use?
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
- “Performance will play a larger role in the second round”
- **Some questions:**
  - What reference platform should we use?
  - Are schemes efficient on small ARMs?
Motivation

- Submission teams focussed on their Intel/AVX2 implementation first
  - As recommended by NIST
- Majority of cryptographic devices is way smaller
  - Limited RAM
  - No/limited vector instructions
  - Side-channels?
- “Performance will play a larger role in the second round”
- Some questions:
  - What reference platform should we use?
  - Are schemes efficient on small ARMs?
  - What are the requirements for RAM and code-size?
“It’s big and it’s slow”
“It’s big and it’s slow”
– everyone, always
Post-quantum on small devices

“*It’s big and it’s slow*”
– everyone, always

- STM32F4DISCOVERY
  - ARM Cortex-M4
  - 32-bit, ARMv7E-M
  - 192 KiB RAM, 168 MHz

- PQM4: test and optimize on the Cortex-M4
  - [github.com/mupq/pqm4](https://github.com/mupq/pqm4)
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1
They are cheap (< $30)

They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1

ARMv7E-M more interesting for assembly optimizations
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1
- ARMv7E-M more interesting for assembly optimizations
- NIST recommended Cortex-M4 for PQC evaluation
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1
- ARMv7E-M more interesting for assembly optimizations
- NIST recommended Cortex-M4 for PQC evaluation
- We’re using it for teaching
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1
- ARMv7E-M more interesting for assembly optimizations
- NIST recommended Cortex-M4 for PQC evaluation
- We’re using it for teaching
  - We have dozens of them lying around
Rationale for using STM32F4DISCOVERY boards

- They are cheap (< $30)
- They are huge
  - You can fit basically everything and then compare
  - Great for PQC – many schemes fit
  - Unfortunately, pqRSA did not terminate within round 1
- ARMv7E-M more interesting for assembly optimizations
- NIST recommended Cortex-M4 for PQC evaluation
- We’re using it for teaching
  - We have dozens of them lying around
  - Our students know how to work with them
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
  - v1: Initial version for round 1
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
  - v1: Initial version for round 1
  - v2: Changes for round 2
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4

- v1: Initial version for round 1
- v2: Changes for round 2
- Status: Recently merged v2; Actively maintained
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   - v1: Initial version for round 1
   - v2: Changes for round 2
   - Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   • v1: Initial version for round 1
   • v2: Changes for round 2
   • Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
   • https://github.com/PQClean/PQClean
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   - v1: Initial version for round 1
   - v2: Changes for round 2
   - Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
   - https://github.com/PQClean/PQClean
   - Status: Framework stable; Integrating schemes now
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   - v1: Initial version for round 1
   - v2: Changes for round 2
   - Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
   - https://github.com/PQClean/PQClean
   - Status: Framework stable; Integrating schemes now

(3) pqriscv: Benchmarking on RISC-V
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   - v1: Initial version for round 1
   - v2: Changes for round 2
   - Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
   - https://github.com/PQClean/PQClean
   - Status: Framework stable; Integrating schemes now

(3) pqriscv: Benchmarking on RISC-V
   - https://github.com/mupq/pqriscv
(1) pqm4: Testing & Benchmarking PQC on the Cortex-M4
   • v1: Initial version for round 1
   • v2: Changes for round 2
   • Status: Recently merged v2; Actively maintained

(2) PQClean: Collecting clean implementations of PQC schemes
   • https://github.com/PQClean/PQClean
   • Status: Framework stable; Integrating schemes now

(3) pqriscv: Benchmarking on RISC-V
   • https://github.com/mupq/pqriscv
   • Status: Very recent; Work in progress
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
- v2 also Ruben Niederhagen, Richard Petri, and Robert Primas
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
  • v2 also Ruben Niederhagen, Richard Petri, and Robert Primas

Originated from PQCRYPTO H2020 EU project
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
  • v2 also Ruben Niederhagen, Richard Petri, and Robert Primas
Originated from PQCRYPTO H2020 EU project
  • March 2015 – February 2018
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

- v2 also Ruben Niederhagen, Richard Petri, and Robert Primas

Originated from PQCRYPTO H2020 EU project

- March 2015 – February 2018
- WP1: post-quantum crypto for small devices
Joint work with Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
  - v2 also Ruben Niederhagen, Richard Petri, and Robert Primas

Originated from PQCRYPTO H2020 EU project
  - March 2015 – February 2018
  - WP1: post-quantum crypto for small devices
  - Deliverable: PQM4 – pq library for the Cortex-M4
Since October 2018 working on ERC project

*Engineering post-quantum cryptography – EPOQUE*
Since October 2018 working on ERC project *Engineering post-quantum cryptography – EPOQUE*

WP1: Secure implementations of post-quantum crypto
Since October 2018 working on ERC project

*Engineering post-quantum cryptography – EPOQUE*

- WP1: Secure implementations of post-quantum crypto
- Build on results of PQCRYPTO, e.g., extend pqm4:
Since October 2018 working on ERC project
*Engineering post-quantum cryptography – EPOQUE*

WP1: Secure implementations of post-quantum crypto

Build on results of PQCRYPTO, e.g., extend pqm4:

- Include more optimized implementations
Since October 2018 working on ERC project
*Engineering post-quantum cryptography – EPOQUE*

WP1: Secure implementations of post-quantum crypto

Build on results of PQCRYPTO, e.g., extend pqm4:

- Include more optimized implementations
- Include implementations with SCA protection
Goals

3 types of implementations

- **ref**: Reference C implementations from submission packages; as little changes as possible
- **opt**: Optimized C implementations; portable
- **m4**: Optimized implementations with parts in ARMv7E-M assembly
Goals

- Framework that eases optimization for this platform
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
- Include as many schemes as possible
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
- Include as many schemes as possible

3 types of implementations
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
- Include as many schemes as possible

3 types of implementations

- ref: Reference C implementations from submission packages; as little changes as possible
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
- Include as many schemes as possible

3 types of implementations

- ref: Reference C implementations from submission packages; as little changes as possible
- opt: Optimized C implementations; portable
Goals

- Framework that eases optimization for this platform
- Automate testing and benchmarking
- Include as many schemes as possible

3 types of implementations

- ref: Reference C implementations from submission packages; as little changes as possible
- opt: Optimized C implementations; portable
- m4: Optimized implementations with parts in ARMv7E-M assembly
Basic functional tests
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies

- Stupid failure tests
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies
- Stupid failure tests
  - KEM: Decaps with different sk doesn’t give the same ss
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies

- Stupid failure tests
  - KEM: Decaps with different sk doesn’t give the same ss
  - KEM: Flipping bits in the ciphertext yields different ss
pqm4: **Functional tests**

- **Basic functional tests**
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies

- **Stupid failure tests**
  - KEM: Decaps with different sk doesn’t give the same ss
  - KEM: Flipping bits in the ciphertext yields different ss
  - Signature: Doesn’t verify with different pk
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies
- Stupid failure tests
  - KEM: Decaps with different sk doesn’t give the same ss
  - KEM: Flipping bits in the ciphertext yields different ss
  - Signature: Doesn’t verify with different pk
- Canaries between keys and messages
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies

- Stupid failure tests
  - KEM: Decaps with different sk doesn’t give the same ss
  - KEM: Flipping bits in the ciphertext yields different ss
  - Signature: Doesn’t verify with different pk

- Canaries between keys and messages
  - Detect writing/reading out of bounds
pqm4: Functional tests

- Basic functional tests
  - KEM: Both parties derive the same key
  - Signature: Valid signature verifies
- Stupid failure tests
  - KEM: Decaps with different sk doesn’t give the same ss
  - KEM: Flipping bits in the ciphertext yields different ss
  - Signature: Doesn’t verify with different pk
- Canaries between keys and messages
  - Detect writing/reading out of bounds
  - Detect assumptions about memory layout (e.g., sk must be after pk – yes, we saw this in practice)
Side-note: How portable is portable C code?

- For various submissions: not very

Common problems

- Assumptions about integer sizes
  - char is not always unsigned
- Floating point arithmetic often leads to problems
  - We had to fix a 25 year old bug in `llrint` of newlib

Many submissions depend on OpenSSL
- This won't work on embedded platforms

Some lessons learned

- Use fixed-sized integers where necessary
- Don't use floating point arithmetic for crypto
- Portable C-code must be tested on different platforms
  - This was part of the motivation for PQClean
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
  - Floating point arithmetic often leads to problems
  - Many submissions depend on OpenSSL
    - This won't work on embedded platforms

Some lessons learned

- Use fixed-sized integers where necessary
- Don’t use floating point arithmetic for crypto
- Portable C-code must be tested on different platforms
  - This was part of the motivation for PQClean
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned

- Many submissions depend on OpenSSL
  - This won’t work on embedded platforms

Some lessons learned:
- Use fixed-sized integers where necessary
- Don’t use floating point arithmetic for crypto
- Portable C-code must be tested on different platforms

This was part of the motivation for PQClean
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned
  - Floating point arithmetic often leads to problems
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in `llrint` of `newlib`
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in \texttt{llrint} of newlib
  - Many submissions depend on OpenSSL
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - `char` is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in `llrint` of newlib
  - Many submissions depend on OpenSSL
    - This won’t work on embedded platforms
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in `llrint` of newlib
  - Many submissions depend on OpenSSL
    - This won’t work on embedded platforms
- Some lessons learned
For various submissions: not very

Common problems

- Assumptions about integer sizes
  - char is not always unsigned
- Floating point arithmetic often leads to problems
  - We had to fix a 25 year old bug in `llrint` of newlib
- Many submissions depend on OpenSSL
  - This won’t work on embedded platforms

Some lessons learned

- Use fixed-sized integers where necessary
For various submissions: not very

Common problems

- Assumptions about integer sizes
  - char is not always unsigned
- Floating point arithmetic often leads to problems
  - We had to fix a 25 year old bug in \texttt{llrint} of newlib
- Many submissions depend on OpenSSL
  - This won’t work on embedded platforms

Some lessons learned

- Use fixed-sized integers where necessary
- Don’t use floating point arithmetic for crypto
Side-note: How portable is portable C code?

- For various submissions: not very

- Common problems
  - Assumptions about integer sizes
    - `char` is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in `llrint` of newlib
  - Many submissions depend on OpenSSL
    - This won’t work on embedded platforms

- Some lessons learned
  - Use fixed-sized integers where necessary
  - Don’t use floating point arithmetic for crypto
  - Portable C-code must be tested on different platforms
Side-note: How portable is portable C code?

- For various submissions: not very
- Common problems
  - Assumptions about integer sizes
    - char is not always unsigned
  - Floating point arithmetic often leads to problems
    - We had to fix a 25 year old bug in `llrint` of newlib
  - Many submissions depend on OpenSSL
    - This won’t work on embedded platforms
- Some lessons learned
  - Use fixed-sized integers where necessary
  - Don’t use floating point arithmetic for crypto
  - Portable C-code must be tested on different platforms
    - This was part of the motivation for PQClean
Goal: Implementations of the same scheme should produce the same testvectors.
Goal: Implementations of the same scheme should produce the same testvectors

- Use deterministic randombytes
Goal: Implementations of the same scheme should produce the same testvectors
  - Use deterministic randombytes

pqm4
Goal: Implementations of the same scheme should produce the same testvectors

- Use deterministic randombytes

pqm4

- Run reference implementations on Intel and M4 and compare
pqm4: Testvector comparison

- **Goal:** Implementations of the same scheme should produce the same testvectors
  - Use deterministic randombytes
- pqm4
  - Run reference implementations on Intel and M4 and compare
  - If optimized implementations are available compare
pqm4: Testvector comparison

- **Goal:** Implementations of the same scheme should produce the same testvectors
  - Use deterministic randombytes
- pqm4
  - Run reference implementations on Intel and M4 and compare
  - If optimized implementations are available compare
- This caught a lot of mistakes
Goal: Implementations of the same scheme should produce the same testvectors

- Use deterministic randombytes

pqm4

- Run reference implementations on Intel and M4 and compare
- If optimized implementations are available compare

This caught a lot of mistakes

It also makes optimizing schemes easier
**Goal:** Implementations of the same scheme should produce the same testvectors

- Use deterministic randombytes

**pqm4**

- Run reference implementations on Intel and M4 and compare
- If optimized implementations are available compare

This caught a lot of mistakes

It also makes optimizing schemes easier

- Compare testvectors during development to catch mistakes early
- **Downside:** Implementations must use randomness in the same way.
- **Downside**: Implementations must use randomness in the same way
  - Example: Storing secret key in a different format is perfectly fine, but won’t give the same testvectors
- **Downside**: Implementations must use randomness in the same way
  - Example: Storing secret key in a different format is perfectly fine, but won’t give the same testvectors
  - Example: Sampling secret key in a different way might be fine, but will produce different testvectors
- **Downside**: Implementations must use randomness in the same way
  - Example: Storing secret key in a different format is perfectly fine, but won’t give the same testvectors
  - Example: Sampling secret key in a different way might be fine, but will produce different testvectors
- We currently don’t have a better way
Fast hashing

1https://github.com/XKCP/XKCP
2https://bench.cr.yp.to/supercop.html
3“All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES

1 https://github.com/XKCP/XKCP
2 https://bench.cr.yp.to/supercop.html
3 “All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES
- We don’t want to benchmark those

---

1 https://github.com/XKCP/XKCP
2 https://bench.cr.yp.to/supercop.html
3 “All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES
- We don’t want to benchmark those
- Our approach: Replace those with a single fast implementation to allow fair comparison

---

1 https://github.com/XKCP/XKCP
2 https://bench.cr.yp.to/supercop.html
3 “All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES
- We don’t want to benchmark those
- Our approach: Replace those with a single fast implementation to allow fair comparison
- SHA-3: ARMv7-M assembly implementation from Keccak team

1. https://github.com/XKCP/XKCP
2. https://bench.cr.yp.to/supercop.html
3. “All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES
- We don’t want to benchmark those
- Our approach: Replace those with a single fast implementation to allow fair comparison
- SHA-3: ARMv7-M assembly implementation from Keccak team
- SHA-2: Fast C-implementation from SUPERCOP

---

1. https://github.com/XKCP/XKCP
2. https://bench.cr.yp.to/supercop.html
3. "All the AES You Need on Cortex-M3 and M4", Schwabe and Stoffelen, SAC 2016
Fast hashing

- Submission packages often come with different implementations of SHA-2, SHA-3, or AES
- We don’t want to benchmark those
- Our approach: Replace those with a single fast implementation to allow fair comparison
- SHA-3: ARMv7-M assembly implementation from Keccak team
- SHA-2: Fast C-implementation from SUPERCOP
- AES: ARMv7-M assembly implementation from [SS16]

1 https://github.com/XKCP/XKCP
2 https://bench.cr.yp.to/supercop.html
3 “All the AES You Need on Cortex-M3 and M4”, Schwabe and Stoffelen, SAC 2016
randombytes
randombytes

- We use the hardware RNG of our platform
randombytes

- We use the hardware RNG of our platform
- Most schemes only sample seed, so speed doesn’t matter
randombytes
- We use the hardware RNG of our platform
- Most schemes only sample seed, so speed doesn’t matter

Cycle counts
randombytes
- We use the hardware RNG of our platform
- Most schemes only sample seed, so speed doesn’t matter

Cycle counts
- We use systick to obtain cycle counts
randombytes
  • We use the hardware RNG of our platform
  • Most schemes only sample seed, so speed doesn’t matter

Cycle counts
  • We use systick to obtain cycle counts
  • We don’t want to benchmark the memory controller
**randombytes**
- We use the hardware RNG of our platform
- Most schemes only sample seed, so speed doesn’t matter

**Cycle counts**
- We use syistick to obtain cycle counts
- We don’t want to benchmark the memory controller
  - Downclock core to 24MHz → no wait states
randombytes
- We use the hardware RNG of our platform
- Most schemes only sample seed, so speed doesn’t matter

Cycle counts
- We use systick to obtain cycle counts
- We don’t want to benchmark the memory controller
  - Downclock core to 24MHz → no wait states
  - Allows to have comprehensible cycle count
Other Benchmarking Challenges

- Compiler versions matter

- When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up.
- To have comparable benchmarks, you need to use the same compiler version!
- Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
  - For pqm4: Use most recent compiler (gcc 8.3.0)

- Compiler flags matter

  - Example: Kyber gets 5% faster when you turn on link time optimisation (-fllto)
  - However, compiler is stupid and dramatically increases stack usage, which needs to be manually fixed
  - pqm4: -O3, no -fllto
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!

- **Compiler flags matter**
  - Example: Kyber gets 5% faster when you turn on link time optimisation (`-flto`)
  - However, compiler is stupid and dramatically increases stack usage, which needs to be manually fixed

For *pqm4*: Use most recent compiler (gcc 8.3.0)
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!
  - Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!
  - Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
  - For pqm4: Use most recent compiler (gcc 8.3.0)
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!
  - Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
  - For pqm4: Use most recent compiler (gcc 8.3.0)

- **Compiler flags matter**
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!
  - Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
  - For pqm4: Use most recent compiler (gcc 8.3.0)

- **Compiler flags matter**
  - Example: Kyber gets 5% faster when you turn on link time optimisation (-flto)
Compiler versions matter

- When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
- To have comparable benchmarks, you need to use the same compiler version!
- Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
- For pqm4: Use most recent compiler (gcc 8.3.0)

Compiler flags matter

- Example: Kyber gets 5% faster when you turn on link time optimisation (-flto)
- However, compiler is stupid and dramatically increases stack usage, which needs to be manually fixed
Other Benchmarking Challenges

- **Compiler versions matter**
  - When upgrading from gcc 5.4.1 to 8.3.0 you can easily get 20% speed-up
  - To have comparable benchmarks, you need to use the same compiler version!
  - Also: Upgrading your compiler can sometimes give you more speed-up than spending weeks writing assembly :(
  - For pqm4: Use most recent compiler (gcc 8.3.0)

- **Compiler flags matter**
  - Example: Kyber gets 5% faster when you turn on link time optimisation (-flto)
  - However, compiler is stupid and dramatically increases stack usage, which needs to be manually fixed
  - pqm4: -O3, no -flto
Other Benchmarking Challenges

- **RAM:** \( 192 \text{ KiB} = 64 \text{ KiB} + 112 \text{ KiB} + 16 \text{ KiB} \)
Other Benchmarking Challenges

- RAM: 192 KiB = 64 KiB + 112 KiB + 16 KiB
  - CCM: 64 KiB mapped at 0x1000 0000 – 0x1000 FFFF

Weird observation: Kyber gets a lot slower when we reduce its stack usage below 16 KiB.

Somehow memory access to memory so SRAM2 is slower.

Benchmarking in “SRAM1 only” gives more stable benchmark results; we’ve changed this in pqm4 v2.
Other Benchmarking Challenges

- RAM: 192 KiB = 64 KiB + 112 KiB + 16 KiB
  - CCM: 64 KiB mapped at 0x1000 0000 – 0x1000 FFFF
  - SRAM1: 112 KiB mapped at 0x20000000 – 0x2001BFFF

- We only use SRAM1+2 for the stack starting at 0x2001FFFF
- Weird observation: Kyber gets a lot slower when we reduce its stack usage below 16 KiB
- Somehow memory access to memory so SRAM2 is slower
- Benchmarking in "SRAM1 only" gives more stable benchmark results; we've changed this in pqm4 v2
Other Benchmarking Challenges

- RAM: 192 KiB = 64 KiB + 112 KiB + 16 KiB
  - CCM: 64 KiB mapped at 0x1000 0000 – 0x1000 FFFF
  - SRAM1: 112 KiB mapped at 0x20000000 – 0x2001BFFF
  - SRAM2: 16 KiB mapped at 0x2001-C000 – 0x2001FFFF
Other Benchmarking Challenges

- RAM: 192 KiB = 64 KiB + 112 KiB + 16 KiB
  - CCM: 64 KiB mapped at 0x1000 0000 – 0x1000 FFFF
  - SRAM1: 112 KiB mapped at 0x20000000 – 0x2001BFFF
  - SRAM2: 16 KiB mapped at 0x2001-C000 – 0x2001FFFF
- We only use SRAM1+2 for the stack starting at 0x2001FFFF

Weird observation: Kyber gets a lot slower when we reduce its stack usage below 16 KiB somehow memory access to memory so SRAM2 is slower

Benchmarking in “SRAM1 only” gives more stable benchmark results; we’ve changed this in pqm4 v2
Other Benchmarking Challenges

- RAM: $192 \text{ KiB} = 64 \text{ KiB} + 112 \text{ KiB} + 16 \text{ KiB}$
  - CCM: $64 \text{ KiB}$ mapped at $0x1000\ 0000 – 0x1000\ FFFF$
  - SRAM1: $112 \text{ KiB}$ mapped at $0x2000\ 0000 – 0x2001\ BFFF$
  - SRAM2: $16 \text{ KiB}$ mapped at $0x2001\ C000 – 0x2001\ FFFF$
- We only use SRAM1+2 for the stack starting at $0x2001\ FFFF$
- Weird observation: Kyber gets a lot slower when we reduce its stack usage below $16 \text{ KiB}$
Other Benchmarking Challenges

- RAM: $192\text{ KiB} = 64\text{ KiB} + 112\text{ KiB} + 16\text{ KiB}$
  - CCM: $64\text{ KiB}$ mapped at $0x1000\ 0000–0x1000\ FFFF$
  - SRAM1: $112\text{ KiB}$ mapped at $0x2000\ 0000–0x2001\ BFFF$
  - SRAM2: $16\text{ KiB}$ mapped at $0x2001\ C000–0x2001\ FFFF$
- We only use SRAM1+2 for the stack starting at $0x2001\ FFFF$
- Weird observation: Kyber gets a lot slower when we reduce its stack usage below $16\text{ KiB}$
- Somehow memory access to memory so SRAM2 is slower
Other Benchmarking Challenges

- RAM: 192 KiB = 64 KiB + 112 KiB + 16 KiB
  - CCM: 64 KiB mapped at 0x1000 0000 – 0x1000 FFFF
  - SRAM1: 112 KiB mapped at 0x20000000 – 0x2001BFFF
  - SRAM2: 16 KiB mapped at 0x2001-C000 – 0x2001FFFF
- We only use SRAM1+2 for the stack starting at 0x2001FFFF
- Weird observation: Kyber gets a lot slower when we reduce its stack usage below 16 KiB
- Somehow memory access to memory so SRAM2 is slower
- Benchmarking in “SRAM1 only” gives more stable benchmark results; we’ve changed this in pqm4 v2
### Signature Schemes

<table>
<thead>
<tr>
<th>scheme</th>
<th>implementation</th>
<th>key generation [cycles]</th>
<th>sign [cycles]</th>
<th>verify [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilithium (100 executions)</td>
<td>m4</td>
<td>AVG: 2,304,135, MIN: 2,303,305, MAX: 2,304,977</td>
<td>AVG: 8,738,743, MIN: 3,118,472, MAX: 32,461,889</td>
<td>AVG: 2,297,215, MIN: 2,296,811, MAX: 2,297,662</td>
</tr>
<tr>
<td>dilithium (100 executions)</td>
<td>ref</td>
<td>AVG: 2,755,209, MIN: 2,754,546, MAX: 2,756,003</td>
<td>AVG: 15,593,609, MIN: 5,001,347, MAX: 47,568,719</td>
<td>AVG: 3,015,099, MIN: 3,014,727, MAX: 3,015,472</td>
</tr>
<tr>
<td>qTesla-I (100 executions)</td>
<td>ref</td>
<td>AVG: 16,181,905, MIN: 7,750,886, MAX: 55,665,081</td>
<td>AVG: 6,528,971, MIN: 1,408,282, MAX: 38,101,487</td>
<td>AVG: 1,038,204, MIN: 1,031,357, MAX: 1,066,661</td>
</tr>
</tbody>
</table>
## Signature Schemes

<table>
<thead>
<tr>
<th>scheme</th>
<th>implementation</th>
<th>key generation [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilithium (100 executions)</td>
<td>m4</td>
<td>AVG: 2,304,135, MIN: 2,303,305, MAX: 2,304,977</td>
</tr>
<tr>
<td>dilithium (100 executions)</td>
<td>ref</td>
<td>AVG: 2,755,209, MIN: 2,754,546, MAX: 2,756,003</td>
</tr>
<tr>
<td>qTesla-l (100 executions)</td>
<td>ref</td>
<td>AVG: 16,181,905, MIN: 7,759,886, MAX: 55,665,081</td>
</tr>
<tr>
<td>Scheme</td>
<td>Implementation</td>
<td>Key Generation [bytes]</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>dilithium</td>
<td>m4</td>
<td>50,896</td>
</tr>
<tr>
<td>dilithium</td>
<td>ref</td>
<td>50,896</td>
</tr>
<tr>
<td>qTesla-l</td>
<td>ref</td>
<td>22,512</td>
</tr>
<tr>
<td>qTesla-III-size</td>
<td>ref</td>
<td>44,016</td>
</tr>
<tr>
<td>qTesla-III-speed</td>
<td>ref</td>
<td>44,024</td>
</tr>
<tr>
<td>sphincs-shake256-128f-simple</td>
<td>clean</td>
<td>2,200</td>
</tr>
</tbody>
</table>
### Signature Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Implementation</th>
<th>Key Generation [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilithium</td>
<td>m4</td>
<td>50,896</td>
</tr>
<tr>
<td>dilithium</td>
<td>ref</td>
<td>50,896</td>
</tr>
<tr>
<td>qTesla-l</td>
<td>ref</td>
<td>22,512</td>
</tr>
</tbody>
</table>
Round 1: Schemes included in pqm4

- See https://github.com/mupq/pqm4/tree/Round1
Round 1: Schemes included in pqm4

- See https://github.com/mupq/pqm4/tree/Round1
- Initially focussed on PQCRIPTO schemes and security level 3
Round 1: Schemes included in pqm4

- See https://github.com/mupq/pqm4/tree/Round1
- Initially focussed on PQCRYPTO schemes and security level 3
- KEMs: FrodoKEM, KINDI, Kyber, NewHope, NTRU-HRSS-KEM, NTRU-KEM, RLizard, Saber, SIKE, NTRUPrime
Round 1: Schemes included in pqm4

- See https://github.com/mupq/pqm4/tree/Round1
- Initially focussed on PQCRYPTO schemes and security level 3
- KEMs: FrodoKEM, KINDI, Kyber, NewHope, NTRU-HRSS-KEM, NTRU-KEM, RLizard, Saber, SIKE, NTRUPrime
- Signature schemes: Dilithium, qTesla, SPHINCS+
Round 1: Schemes included in pqm4

- See https://github.com/mupq/pqm4/tree/Round1
- Initially focussed on PQCRYPTO schemes and security level 3
- KEMs: FrodoKEM, KINDI, Kyber, NewHope, NTRU-HRSS-KEM, NTRU-KEM, NTRU, RLizard, Saber, SIKE, NTRUPrime
- Signature schemes: Dilithium, qTesla, SPHINCS+
Papers using pqm4

- “Saber on ARM CCA-secure module lattice-based key encapsulation on ARM”, Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede, CHES’18 (ePrint 2018/682)
Papers using pqm4

- “Saber on ARM CCA-secure module lattice-based key encapsulation on ARM”, Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede, CHES’18 (ePrint 2018/682)
  - Optimising speed and stack usage of Saber for M4+M0
Papers using pqm4

- “Saber on ARM CCA-secure module lattice-based key encapsulation on ARM”, Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede, CHES’18 (ePrint 2018/682)
  - Optimising speed and stack usage of Saber for M4+M0
- “Faster multiplication in $\mathbb{Z}_{2^m} [x]$ on Cortex-M4 to speed up NIST PQC candidates”, Kannwischer, Rijneveld, and Schwabe, ACNS’19 (ePrint 2018/1018)
Papers using pqm4

- “Saber on ARM CCA-secure module lattice-based key encapsulation on ARM”, Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede, CHES’18 (ePrint 2018/682)
  - Optimising speed and stack usage of Saber for M4+M0

- “Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates”, Kannwischer, Rijneveld, and Schwabe, ACNS’19 (ePrint 2018/1018)
  - Optimizing Kindi, NTRUEncrypt, NTRU-HRSS, RLizard, and Saber for speed
Papers using pqm4 (2)

- “Evaluation of Lattice-Based Signature Schemes in Embedded Systems”, Güneysu, Krausz, Oder, and Speith, ICECS’18, (link to paper)
Papers using pqm4 (2)

- “Evaluation of Lattice-Based Signature Schemes in Embedded Systems”, Güneysu, Krausz, Oder, and Speith, ICECS’18, (link to paper)
  - Optimising GLP, BLISS, and Dilithium for the M4
“Evaluation of Lattice-Based Signature Schemes in Embedded Systems”, Güneysu, Krausz, Oder, and Speith, ICECS’18, (link to paper)
- Optimising GLP, BLISS, and Dilithium for the M4

“Towards Practical Microcontroller Implementation of the Signature Scheme Falcon” Oder, Speith, Höltingen, and Güneysu, PQCrypto’19 (link to paper)
Papers using pqm4 (2)

- “Evaluation of Lattice-Based Signature Schemes in Embedded Systems”, Güneysu, Krausz, Oder, and Speith, ICECS’18, (link to paper)
  - Optimising GLP, BLISS, and Dilithium for the M4
- “Towards Practical Microcontroller Implementation of the Signature Scheme Falcon” Oder, Speith, Höltgen, and Güneysu, PQCrypto’19 (link to paper)
  - Optimising Falcon for the M4
“Number “Not Used” Once - Practical fault attack on pqm4 implementations of NIST candidates”, Ravi, Basu Roy, Bhasin, Chattopadhyay, and Mukhopadhyay, COSADE’19 (ePrint 2018/211)
“Number “Not Used” Once - Practical fault attack on \textit{pqm4 implementations of NIST candidates}”, Ravi, Basu Roy, Bhasin, Chattopadhyay, and Mukhopadhyay, COSADE’19 (ePrint 2018/211)

- EM induced instruction skipping on reference implementations that cause secret and error to be the same in LWE-based schemes
  (validated on FrodoKEM, NewHope, Kyber, Dilithium)
Papers using \texttt{pqm4} (3)

- “Number “Not Used” Once - Practical fault attack on \texttt{pqm4 implementations of NIST candidates}”, Ravi, Basu Roy, Bhasin, Chattopadhyay, and Mukhopadhyay, COSADE’19 (ePrint 2018/211)
  - EM induced instruction skipping on reference implementations that cause secret and error to be the same in LWE-based schemes (validated on FrodoKEM, NewHope, Kyber, Dilithium)
  - IMHO: Unrelated to \texttt{pqm4}; we don’t claim any fault or side-channel resistance
Papers using pqm4 (4)

- “Fly, you fool! Faster Frodo for the ARM Cortex-M4”, Bos, Friedberger, Martinoli, Oswald, and Stam (ePrint 2018/1116)
Papers using pqm4 (4)

- “Fly, you fool! Faster Frodo for the ARM Cortex-M4”, Bos, Friedberger, Martinoli, Oswald, and Stam (ePrint 2018/1116)
  - Optimising Frodo for the M4
Papers using pqm4 (4)

- “Fly, you fool! Faster Frodo for the ARM Cortex-M4”, Bos, Friedberger, Martinoli, Oswald, and Stam (ePrint 2018/1116)
  - Optimising Frodo for the M4

- “Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4”, Botros, Kannwischer, and Schwabe (soon on ePrint)
Papers using pqm4 (4)

- “Fly, you fool! Faster Frodo for the ARM Cortex-M4”, Bos, Friedberger, Martinoli, Oswald, and Stam (ePrint 2018/1116)
  - Optimising Frodo for the M4
- “Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4”, Botros, Kannwischer, and Schwabe (soon on ePrint)
  - Optimising speed and stack usage of Kyber for the Cortex-M4
Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)

- Spoiler: A lot for most schemes
Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
  • Spoiler: A lot for most schemes

Measure code-size
- Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
  - Spoiler: A lot for most schemes
- Measure code-size
- Benchmarking in SRAM1 to get more sensible cycle counts
Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
  * Spoiler: A lot for most schemes

- Measure code-size
- Benchmarking in SRAM1 to get more sensible cycle counts
- Simplified build process
- Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
  - Spoiler: A lot for most schemes
- Measure code-size
- Benchmarking in SRAM1 to get more sensible cycle counts
- Simplified build process
- Hardware abstraction to allow benchmarking on other platforms (see pqriscv)
- Benchmark how much time is spent in symmetric primitives (SHA-2, SHA-3, AES)
  - Spoiler: A lot for most schemes
- Measure code-size
- Benchmarking in SRAM1 to get more sensible cycle counts
- Simplified build process
- Hardware abstraction to allow benchmarking on other platforms (see pqriscv)
- Integration of PQClean
# Signature Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Implementation</th>
<th>Key Generation [%]</th>
<th>Sign [%]</th>
<th>Verify [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilithium</td>
<td>m4</td>
<td>73.4%</td>
<td>42.3%</td>
<td>66.7%</td>
</tr>
<tr>
<td>dilithium</td>
<td>ref</td>
<td>61.4%</td>
<td>25.4%</td>
<td>50.8%</td>
</tr>
<tr>
<td>qTesla-I</td>
<td>ref</td>
<td>63.5%</td>
<td>27.8%</td>
<td>32.4%</td>
</tr>
<tr>
<td>qTesla-III-size</td>
<td>ref</td>
<td>66.2%</td>
<td>23.0%</td>
<td>27.5%</td>
</tr>
<tr>
<td>qTesla-III-speed</td>
<td>ref</td>
<td>73.3%</td>
<td>24.1%</td>
<td>28.3%</td>
</tr>
<tr>
<td>sphincs-shake256-128f-simple</td>
<td>clean</td>
<td>96.4%</td>
<td>96.2%</td>
<td>99.1%</td>
</tr>
<tr>
<td>Scheme</td>
<td>Implementation</td>
<td>Key Generation [%]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dilithium</td>
<td>m4</td>
<td>73.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dilithium</td>
<td>ref</td>
<td>61.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qTesla-I</td>
<td>ref</td>
<td>63.5%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Signature Schemes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>diiithium</td>
<td>m4</td>
<td>14,864</td>
<td>0</td>
<td>0</td>
<td>14,864</td>
</tr>
<tr>
<td>diiithium</td>
<td>ref</td>
<td>9,788</td>
<td>0</td>
<td>0</td>
<td>9,788</td>
</tr>
<tr>
<td>qTesla-I</td>
<td>ref</td>
<td>16,176</td>
<td>0</td>
<td>0</td>
<td>16,176</td>
</tr>
<tr>
<td>qTesla-III-size</td>
<td>ref</td>
<td>24,804</td>
<td>0</td>
<td>0</td>
<td>24,804</td>
</tr>
<tr>
<td>qTesla-III-speed</td>
<td>ref</td>
<td>23,716</td>
<td>0</td>
<td>0</td>
<td>23,716</td>
</tr>
<tr>
<td>sphincs-shake256-128f-simple</td>
<td>clean</td>
<td>3,932</td>
<td>0</td>
<td>0</td>
<td>3,932</td>
</tr>
</tbody>
</table>
**pqm4 Benchmarks: Code-size**

## Signature Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Implementation</th>
<th>.text [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilithium</td>
<td>m4</td>
<td>14,864</td>
</tr>
<tr>
<td>dilithium</td>
<td>ref</td>
<td>9,788</td>
</tr>
<tr>
<td>qTesla-I</td>
<td>ref</td>
<td>16,176</td>
</tr>
</tbody>
</table>
PQClean
**PQClean**

- https://github.com/PQClean/PQClean

Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers

**Goal**: Provide "clean" C implementations of 2nd round NIST PQC candidates

- Easily integrable into other projects

Still a lot of work in progress

Currently includes:
- FrodoKEM (6x), Kyber (3x), MQDSS (2x), SPHINCS(36x), NTRU (4x)
https://github.com/PQClean/PQClean
Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers
Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers

Goal: Provide “clean” C implementations of 2nd round NIST PQC candidates
Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers

**Goal:** Provide “clean” C implementations of 2nd round NIST PQC candidates
  - Easily integrable into other projects

https://github.com/PQClean/PQClean
Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers

**Goal**: Provide “clean” C implementations of 2nd round NIST PQC candidates
  - Easily integrable into other projects

Still a lot of work in progress
PQClean

- [link](https://github.com/PQClean/PQClean)
- Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers
- **Goal:** Provide “clean” C implementations of 2nd round NIST PQC candidates
  - Easily integrable into other projects
- Still a lot of work in progress
- Currently includes:
  - FrodoKEM (6x), Kyber (3x), MQDSS (2x), SPHINCS(36x), NTRU (4x)
PQClean

- [GitHub Link](https://github.com/PQClean/PQClean)
- Joint work with Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers
- **Goal**: Provide “clean” C implementations of 2nd round NIST PQC candidates
  - Easily integrable into other projects
- Still a lot of work in progress
- Currently includes:
  - FrodoKEM (6x), Kyber (3x), MQDSS (2x), SPHINCS(36x), NTRU (4x)
PQClean: Code requirements

- Code is valid C99
Code is valid C99

Compiles with -Wall -Wextra -Wpedantic -Werror -Wmissing-prototypes with gcc and clang
PQClean: Code requirements

- Code is valid C99
- Compiles with `-Wall -Wextra -Wpedantic -Werror -Wmissing-prototypes` with gcc and clang
- `#if/#ifdefs` only for header encapsulation
PQClean: Code requirements

- Code is valid C99
- Compiles with `-Wall -Wextra -Wpedantic -Werror -Wmissing-prototypes` with gcc and clang
- `#if/#ifdefs` only for header encapsulation
- Only dependencies: `fips202.c, sha2.c, aes.c, randombytes.c`
PQClean: Code requirements

- Code is valid C99
- Compiles with -Wall -Wextra -Wpedantic -Werror -Wmissing-prototypes with gcc and clang
- #if/#ifndefs only for header encapsulation
- Only dependencies: fips202.c, sha2.c, aes.c, randombytes.c
- No dynamic memory allocations (including variable-length arrays)
Output-parameter pointers in functions are on the left
- Output-parameter pointers in functions are on the left
- const arguments are labeled as const
- Output-parameter pointers in functions are on the left
- const arguments are labeled as const
- All exported symbols are namespaced in place
Output-parameter pointers in functions are on the left
- const arguments are labeled as const
- All exported symbols are namespaced in place
- Integer types are of fixed size where relevant
Output-parameter pointers in functions are on the left
- const arguments are labeled as `const`
- All exported symbols are namespaced in place
- Integer types are of fixed size where relevant
- Integers used for indexing memory are of size `size_t`
- Output-parameter pointers in functions are on the left
- `const` arguments are labeled as `const`
- All exported symbols are namespaced in place
- Integer types are of fixed size where relevant
- Integers used for indexing memory are of size `size_t`
- Variable declarations at the beginning
- Output-parameter pointers in functions are on the left
- `const` arguments are labeled as `const`
- All exported symbols are namespaced in place
- Integer types are of fixed size where relevant
- Integers used for indexing memory are of size `size_t`
- Variable declarations at the beginning
  - Except in `for(size_t i=...)`
PQClean: Code requirements (3)

- Passes functional tests
- Passes functional tests
- API functions do not write outside provided buffers
- Passes functional tests
- API functions do not write outside provided buffers
- Consistent test vectors across runs
PQClean: Code requirements (3)

- Passes functional tests
- API functions do not write outside provided buffers
- Consistent test vectors across runs
- Consistent test vectors on big- and little-endian machines
PQClean: Code requirements (3)

- Passes functional tests
- API functions do not write outside provided buffers
- Consistent test vectors across runs
- Consistent test vectors on big- and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines
- Passes functional tests
- API functions do not write outside provided buffers
- Consistent test vectors across runs
- Consistent test vectors on big- and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines
- No errors/warnings reported by valgrind
PQClean: Code requirements (3)

- Passes functional tests
- API functions do not write outside provided buffers
- Consistent test vectors across runs
- Consistent test vectors on big- and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines
- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
Most of these requirements are checked automatically by CI.
Most of these requirements are checked automatically by CI

- Linux, OS X, and Windows
Most of these requirements are checked automatically by CI

- Linux, OS X, and Windows
- gcc and clang
Most of these requirements are checked automatically by CI

- Linux, OS X, and Windows
- gcc and clang
- 32-bit and 64-bit
Most of these requirements are checked automatically by CI

- Linux, OS X, and Windows
- gcc and clang
- 32-bit and 64-bit
- Intel, ARM, and PowerPC
Integrations

- Open Quantum Safe ¹:

¹https://github.com/open-quantum-safe/
²https://github.com/mupq/pqm4
³https://github.com/mupq/pqriscv
Integrations

- Open Quantum Safe\(^1\):
  - Integrates into forks of OpenSSL and OpenSSH

1\url{https://github.com/open-quantum-safe/}
2\url{https://github.com/mupq/pqm4}
3\url{https://github.com/mupq/pqriscv}
Open Quantum Safe\(^1\):
- Integrates into forks of OpenSSL and OpenSSH

pqm4\(^2\):

\(^1\)https://github.com/open-quantum-safe/
\(^2\)https://github.com/mupq/pqm4
\(^3\)https://github.com/mupq/pqriscv
Integrations

- **Open Quantum Safe** \(^1\):
  - Integrates into forks of OpenSSL and OpenSSH
- **pqm4** \(^2\):
  - All PQClean schemes that fit in 112 KiB of RAM are automatically benchmarked

\(^1\)https://github.com/open-quantum-safe/
\(^2\)https://github.com/mupq/pqm4
\(^3\)https://github.com/mupq/pqriscv
Integrations

- Open Quantum Safe \(^1\):
  - Integrates into forks of OpenSSL and OpenSSH
- pqm4 \(^2\):
  - All PQClean schemes that fit in 112 KiB of RAM are automatically benchmarked
- pqriscv \(^3\):

\(^1\)https://github.com/open-quantum-safe/
\(^2\)https://github.com/mupq/pqm4
\(^3\)https://github.com/mupq/pqriscv
Integrations

- Open Quantum Safe ¹:
  - Integrates into forks of OpenSSL and OpenSSH
- pqm4 ²:
  - All PQClean schemes that fit in 112 KiB of RAM are automatically benchmarked
- pqriscv ³:
  - All PQClean schemes will be benchmarked on RISC-V in the future

¹https://github.com/open-quantum-safe/
²https://github.com/mupq/pqm4
³https://github.com/mupq/pqriscv
pqriscv
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

1 https://github.com/mupq/mupq
2 https://github.com/SpinalHDL/VexRiscv
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

Work on this just started – no results yet

---

1https://github.com/mupq/mupq
2https://github.com/SpinalHDL/VexRiscv
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

Work on this just started – no results yet

Similar motivation as pqm4; common code in mupq\textsuperscript{1}

\textsuperscript{1}https://github.com/mupq/mupq

\textsuperscript{2}https://github.com/SpinalHDL/VexRiscv
- Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
- Work on this just started – no results yet
- Similar motivation as pqm4; common code in mupq\(^1\)
- Using the VexRiscv Core\(^2\)

\(^1\)https://github.com/mupq/mupq
\(^2\)https://github.com/SpinalHDL/VexRiscv
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

Work on this just started – no results yet

Similar motivation as pqm4; common code in mupq\textsuperscript{1}

Using the VexRiscv Core\textsuperscript{2}

- Tiny if it has to be (1399 LUT 971 FF @ Artix-7 with single cycle multiplier & barrel shifter)

\textsuperscript{1}\url{https://github.com/mupq/mupq}
\textsuperscript{2}\url{https://github.com/SpinalHDL/VexRiscv}
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

Work on this just started – no results yet

Similar motivation as pqm4; common code in mupq\textsuperscript{1}

Using the VexRiscv Core\textsuperscript{2}

- Tiny if it has to be (1399 LUT 971 FF @ Artix-7 with single cycle multiplier & barrel shifter)
- Powerful if necessary (Caches, MMU, pipeline stage bypass, ...)

\textsuperscript{1}https://github.com/mupq/mupq
\textsuperscript{2}https://github.com/SpinalHDL/VexRiscv
• Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen
• Work on this just started – no results yet
• Similar motivation as pqm4; common code in mupq\(^1\)
• Using the VexRiscv Core\(^2\)
  • Tiny if it has to be (1399 LUT 971 FF @ Artix-7 with single cycle multiplier & barrel shifter)
  • Powerful if necessary (Caches, MMU, pipeline stage bypass, ...)
  • Extremely flexible, any 5 stages easily extendable

\(^1\)https://github.com/mupq/mupq
\(^2\)https://github.com/SpinalHDL/VexRiscv
Joint work with Ruben Niederhagen, Richard Petri, Robert Primas, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen

Work on this just started – no results yet

Similar motivation as pqm4; common code in mupq\(^1\)

Using the VexRiscv Core\(^2\)

- Tiny if it has to be (1399 LUT 971 FF @ Artix-7 with single cycle multiplier & barrel shifter)
- Powerful if necessary (Caches, MMU, pipeline stage bypass, ...)
- Extremely flexible, any 5 stages easily extendable
- Cycle accurate simulation (including debugger!)

\(^1\)https://github.com/mupq/mupq
\(^2\)https://github.com/SpinalHDL/VexRiscv
Possible alternative platform for RISC-V, easier for experimental extensions
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment

Once we have a baseline reference, we can start more experiments
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment

Once we have a baseline reference, we can start more experiments
  - Instruction set extensions
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment

Once we have a baseline reference, we can start more experiments

- Instruction set extensions
- Accelerators
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment

Once we have a baseline reference, we can start more experiments
  - Instruction set extensions
  - Accelerators
  - Co-processors applicable to multiple schemes
Possible alternative platform for RISC-V, easier for experimental extensions

Automatic benchmarks in a CI environment

Once we have a baseline reference, we can start more experiments

- Instruction set extensions
- Accelerators
- Co-processors applicable to multiple schemes
- SCA Countermeasures
Contributing to PQClean, pqm4, and pqriscv

- Contribute clean C reference implementations to PQClean\(^1\)

---

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
Contribute clean C reference implementations to PQClean\(^1\)
- They will be benchmarked on the pqm4 and pqriscv

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
Contribute clean C reference implementations to PQClean\(^1\)
- They will be benchmarked on the pqm4 and pqriscv
- Functional reference for platform specific optimizations

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
Contributing to PQClean, pqm4, and pqriscv

- Contribute clean C reference implementations to PQClean¹
  - They will be benchmarked on the pqm4 and pqriscv
  - Functional reference for platform specific optimizations
- Goal: Have all schemes in by the end of May

¹https://github.com/PQClean/PQClean
²https://github.com/mupq/mupq
³https://github.com/mupq/pqm4
⁴https://github.com/mupq/pqriscv
Contributing to PQClean, pqm4, and pqriscv

- Contribute clean C reference implementations to PQClean\(^1\)
  - They will be benchmarked on the pqm4 and pqriscv
  - Functional reference for platform specific optimizations
  - **Goal:** Have all schemes in by the end of May

- Contribute optimized C implementations to mupq\(^2\)

---

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
Contributing to PQClean, pqm4, and pqriscv

- Contribute clean C reference implementations to PQClean\(^1\)
  - They will be benchmarked on the pqm4 and pqriscv
  - Functional reference for platform specific optimizations
  - **Goal:** Have all schemes in by the end of May

- Contribute optimized C implementations to mupq\(^2\)
  - They will be benchmarked on the pqm4 and pqriscv

---

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
Contributing to PQClean, pqm4, and pqriscv

- Contribute clean C reference implementations to PQClean\(^1\)
  - They will be benchmarked on the pqm4 and pqriscv
  - Functional reference for platform specific optimizations
  - **Goal:** Have all schemes in by the end of May

- Contribute optimized C implementations to mupq\(^2\)
  - They will be benchmarked on the pqm4 and pqriscv

- Contribute optimized assembly implementations to pqm4\(^3\) and pqriscv\(^4\)

\(^1\)https://github.com/PQClean/PQClean
\(^2\)https://github.com/mupq/mupq
\(^3\)https://github.com/mupq/pqm4
\(^4\)https://github.com/mupq/pqriscv
https://github.com/mupq/pqm4
https://github.com/mupq/pqriscv
https://github.com/PQClean/PQClean

slides available at kannwischer.eu

Thank you!